DEGREE PROJECT, IN MASTER'S PROGRAMME, SYSTEM-ON-CHIP DESIGN (TSKKM) , SECOND LEVEL

*STOCKHOLM*, *SWEDEN 2015*

# TLM-RTL Equivalence Automated Verification Method and Tool Evaluation

## PERFORMED IN ERICSSON AB

CHENCHENG ZHANG

**KTH ROYAL INSTITUTE OF TECHNOLOGY**

**SCHOOL OF INFORMATION AND COMMUNICATION TECHNOLOGY**

TRITA TRITA-ICT-EX-2015:6

# Abstract

Raising the abstraction level, from Register Transfer Level (RTL) to Transaction Level Model (TLM), accelerates the design and simulation speed for Application Specific Integrated Circuit (ASIC) design in modern industry.

TLM can be used for reference model for RTL in verification process. TLM can also be used in High Level Synthesis (HLS) as the original model for RTL implementation. With the development of design automation tools for HLS by big Electronic Design Automation (EDA) vendors in recent years, it is now possible to generate RTL code from TLM directly by these fully automated industry commercial tools.

When it comes up to automated ASIC design, the functional consistency between TLM and RTL is the top task of the verification. Several methodologies are developed or under development in both industry and research area. Some automated verification tools may help to greatly reduce the verification effort.

This thesis report has proposed a flow for verifying the functional equivalence between TLM/SystemC/C code against its corresponding RTL code. The RTL code is the result of automated HLS from TLM. One commercial tool based on sequential equivalence checking (SEC) has been evaluated and its limitations have been assessed. The way to overcome those limitations by complementary verification methods has been scrutinized.

# Acknowledgement

I will firstly give my deepest and biggest gratitude to my supervisor, Björn Fjellborg in Digital ASIC Department of Ericsson. He gave me this chance to do the thesis research in Ericsson and helped me to get a deep understanding on functional verification process in modern industry. I benefitted a lot from his instructions throughout the thesis work.

Secondly, I will express thanks to Richard Langridge from Calypto. He was my technical support of using the SLEC tool. We worked together on testing the tool and conquer many bugs. It would be extremely hard without Richard to make myself understand this tool.

Third, I will also show my gratitude to my examiner, Zhonghai Lu, in KTH. He helped me in the research area and gave a lot of courage to me. We have made several pleasant discussions which made my view on the thesis much clearer.

I would also like to show thankfulness to my manager Pierre Rohdin who also gave me the chance to perform this thesis work in Ericsson. He also helped me a lot during the starting phase.

Thousands of thanks will be given to my colleague, Amir Namvar, who was the former thesis worker on the previous project. He made me understand on one of the ASIC design in Ericsson and helped me a lot on using the automated HLS vendor tools.

I will also express gratitude to Frederic Pouyet, who gave me several perfect lessons on some basic verification methodologies.

Same appreciations to my colleagues: Arjun Yedurupaka, Xinwei Xu, Shuxiang Xu, and Lei Liang. You all gave me a lot of help on many technical problems.

Finally, thanks to my parents, thanks to my roommate, and thanks to all the colleagues including Sheng yee Pang, Dilip Kumar Vajravelu, Pratyaksha Navalkar Ravindra, Noora Solati, Shaghayegh Tabatabaei, Shafqat Ullah, Sujatha Rajagopal, Dandan Yan, Gazizeh Babataheri, Michele Barboncini, Fan Yang, Bibin Babu, Orri Tomasson, Ioannis Savvidis, and Alberto Saggio, who accompanied me and gave me a lot of happy time during my thesis work in Ericsson.

## Terminology

| | |
|---|---|
| ABV | Assertion Based Verification |
| ANSI | American National Standards Institute |
| ASIC | Application Specific Integrated Circuit |
| AVM | Advanced Verification Methodology |
| CEX | Counter Example |
| CPU | Central Processing Unit |
| DSP | Digital Signal Processor |
| DUT | Device under Test |
| EDA | Electronic Design Automation |
| ESL | Electronic System Level |
| FPGA | Field Programmable Gate Array |
| GPU | Graphics Processing Unit |
| GUI | Graphical User Interface |
| HECTOR | High-level Equivalence C TO RTL |
| HLS | High-level Synthesis |
| HW | Hardware |
| IP | Intellectual Property |
| MUX | Multiplexer |
| OOBA | Out of Boundary Access |

| | |
|---|---|
| OVM | Open Verification Methodology |
| PSL | Property Specification Language |
| RBS | Radio Base Station |
| RTL | Register-transfer Level |
| SC | SystemC |
| SEC | Sequential Equivalence Checking |
| SLEC | Sequential Logic Equivalence Checker |
| SLM | System-level Model |
| SoC | System-on-chip |
| SV | SystemVerilog |
| SVA | SystemVerilog Assertion |
| SW | Software |
| TLM | Transaction-level Modeling |
| UVC | UVM Verification Component |
| UVM | Universal Verification Methodology |
| VCD | Value Change Dump |
| VIP | Verification IP |

# Contents

**ABSTRACT** ..... **2**

**ACKNOWLEDGEMENT** ..... **3**

**TERMINOLOGY** ..... **4**

**CONTENTS** ..... **6**

**LIST OF FIGURES** ..... **8**

**LIST OF TABLES** ..... **8**

**1 INTRODUCTION** ..... **9**

1.1 BACKGROUND ..... 9
*1.1.1 SoC Design Flow* ..... *11*
*1.1.2 SoC & ASIC Design at Ericsson* ..... *13*
1.2 PROBLEM ..... 16
1.3 MOTIVATION AND GOAL ..... 16

**2 VERIFICATION METHOD** ..... **17**

2.1 UNIVERSAL VERIFICATION METHODOLOGY ..... 18
2.2 METHODS STILL UNDER RESEARCH ..... 21
*2.2.1 TLM Verification* ..... *21*
*2.2.2 Assertion-based TLM to RTL Verification Method with Reuse Tool* ..... *22*
*2.2.3 Verification Tools without Testbench* ..... *23*
2.3 TLM-RTL VERIFICATION TOOLS WITHOUT TESTBENCH ..... 25
*2.3.1 HECTOR* ..... *25*
*2.3.2 SLEC* ..... *26*
*2.3.3 Comparison* ..... *26*

**3 VERIFICATION USING SLEC** ..... **28**

3.1 PRINCIPLE ..... 28
3.2 VERIFICATION PLAN ..... 32
3.3 SLEC VERIFICATION FLOW ..... 34
3.4 VERIFICATION EXAMPLES ..... 41
*3.4.1 Ericsson DERM Block* ..... *42*
3.4.1.1 Code Structure ..... 42
3.4.1.2 Verification using SLEC ..... 44
3.4.1.3 Summary ..... 46
*3.4.2 Simple Transaction Block with Memory via Channel* ..... *48*
3.4.2.1 Code Structure ..... 48
3.4.2.2 Verification using SLEC ..... 50
3.4.2.3 Summary ..... 51

**4 CONCLUSION** ..... **52**

4.1 ACHIEVEMENTS ..... 52
4.2 FUTURE WORK ..... 54

**5 SUPPLEMENTARY NOTES** ..... **55**

**6 REFERENCES** ..... **56**

**7 APPENDIX** ..... **58**

7.1 SIMPLE TRANSACTION BLOCK WITH MEMORY VIA CHANNEL ..... 58
*7.1.1 C Code* ..... *58*
*7.1.2 Catapult SL Command Script* ..... *60*

*7.1.3 Custom Constraints Settings ..... 62*
*7.1.4 SLEC Result ..... 63*

# List of Figures

FIGURE 1 COMPUTING GROWTH DRIVERS (SOURCE: MORGAN STANLEY RESEARCH) ..... 10
FIGURE 2 THE GROWING COST OF VERIFICATION (SOURCE: SYNOPSYS) ..... 11
FIGURE 3 TYPICAL SOC DESIGN FLOW ..... 12
FIGURE 4 ASIC DESIGN FLOW (SOURCE: ERICSSON) ..... 13
FIGURE 5 RTL FUNCTIONAL VERIFICATION FLOW (SOURCE: ERICSSON) ..... 14
FIGURE 6 USING HLS TO AUTOMATE DESIGN FLOW ..... 15
FIGURE 7 HISTORY OF VERIFICATION METHODOLOGIES ..... 18
FIGURE 8 UVM HIERARCHY ..... 19
FIGURE 9 BMC BASED UNTIMED TLM PROPERTY CHECK (SOURCE: [10]) ..... 21
FIGURE 10 SVM STRUCTURE (SOURCE: [12]) ..... 22
FIGURE 11 VERT CONSISTENCY CHECKING FRAMEWORK (SOURCE: [13]) ..... 23
FIGURE 12 HECTOR VERIFICATION FLOW ..... 26
FIGURE 13 SIMPLE PARALLEL ADDER ..... 29
FIGURE 14 SIMPLE SERIAL ADDER ..... 29
FIGURE 15 UNIT PRODUCT MACHINE FOR SIMPLE ADDER ..... 30
FIGURE 16 SIMPLIFY THE VERIFICATION FLOW ..... 32
FIGURE 17 SLEC-CATAPULT DESIGN FLOW ..... 35
FIGURE 18 ENABLE SLEC FLOW IN CATAPULT ..... 36
FIGURE 19 CAT2SLEC WRAPPER EXAMPLE ..... 37
FIGURE 20 SIMPLE TRANSACTION BLOCK ..... 48
FIGURE 21 OVERVIEW OF NEW VERIFICATION FLOW ..... 53

# List of Tables

TABLE 1 SLEC DESIGN LIBRARY ..... 35
TABLE 2 OPTIONS TO CUSTOMIZE CAT2SLEC FLOW ..... 39
TABLE 3 FILE STRUCTURE OF SLEC VERIFICATION FLOW ENVIRONMENT ..... 40

# 1 Introduction

Nowadays, Transaction-level Modeling (TLM) [1] has been introduced to system design. This concept enables a higher level of abstraction of models than Register-transfer Level (RTL) with less detail description on Hardware (HW). TLM simplifies and accelerates the design process by focusing on the pure functionality of design. It removes the temporal and HW structural details and abstracts the design into transactions between functional blocks. This overcomes the limitation of RTL-driven design which requires every single micro-architectural detail.

With the development of TLM, designers nowadays are able to design and verify system models in different abstraction levels. TLM can even be the reference model for further RTL design or synthesis. This greatly improves the design automation by bring forward automated step from logic synthesis to High-level Synthesis (HLS).

Verification between TLM and RTL has become of most importance in modern HLS design flow. The consistence of functionality has to be validated for both TLM and RTL. However, verification for RTL models separately is still required. This is because not only the algorithms of those functional implementations in TLM have to be validated, but also those micro-architectures in RTL should match the specified requirements.

## 1.1 Background

In recent years, chips with more functional capabilities, higher performances, as well as smaller scale are highly demanded in all kinds of appliances, such as smartphones, tablets, car electronics, home entertainment devices, etc. The following figure shows the trend of growth for the computing devices:

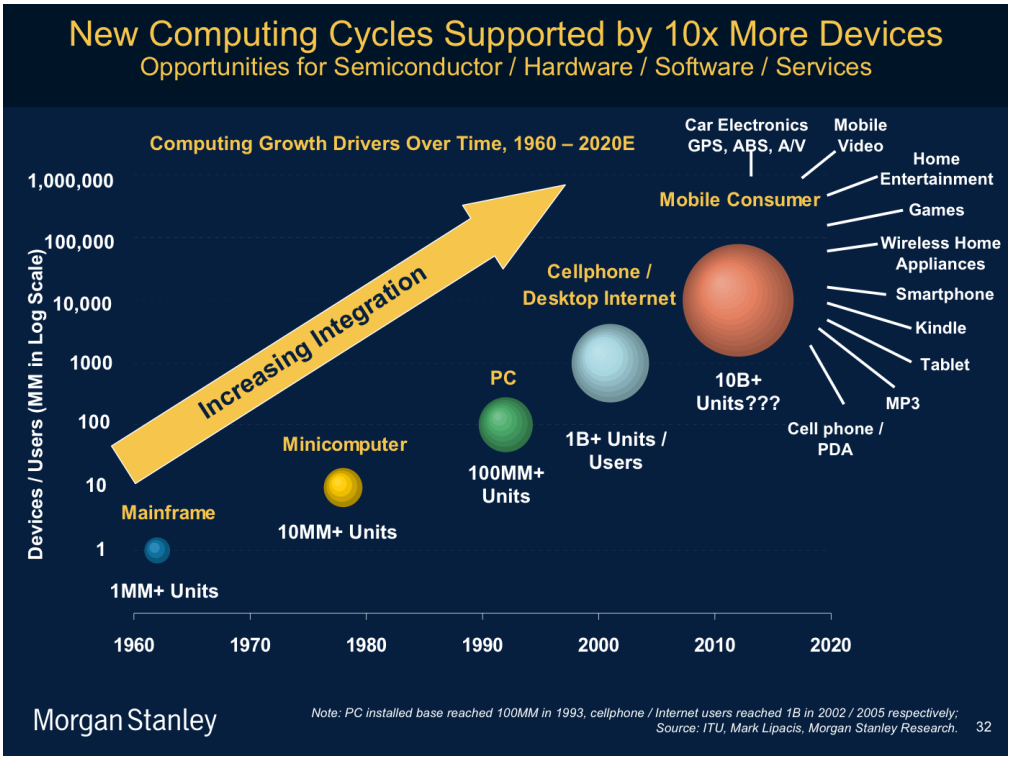


**Figure 1 Computing Growth Drivers (Source: Morgan Stanley Research)**

This trend causes the rapid growing of system-on-chip (SoC) complexity. More and more functionality, such as CPU, GPU, Wireless Processor, etc., are integrated into SoCs. Correspondingly, the design effort and verification effort keeps on increasing.

It brings up the problem that both design and verification efforts will become very massive and require huge amount of cost. Furthermore, in a complete SoC design flow, the percentage of verification and validation effort is much higher than those design effort. Verification and validation effort includes simulations, assertions, formal checking, coverage checking, and debugging. The following figure indicates the trend of cost growth of industrial verification effort during the past decade.

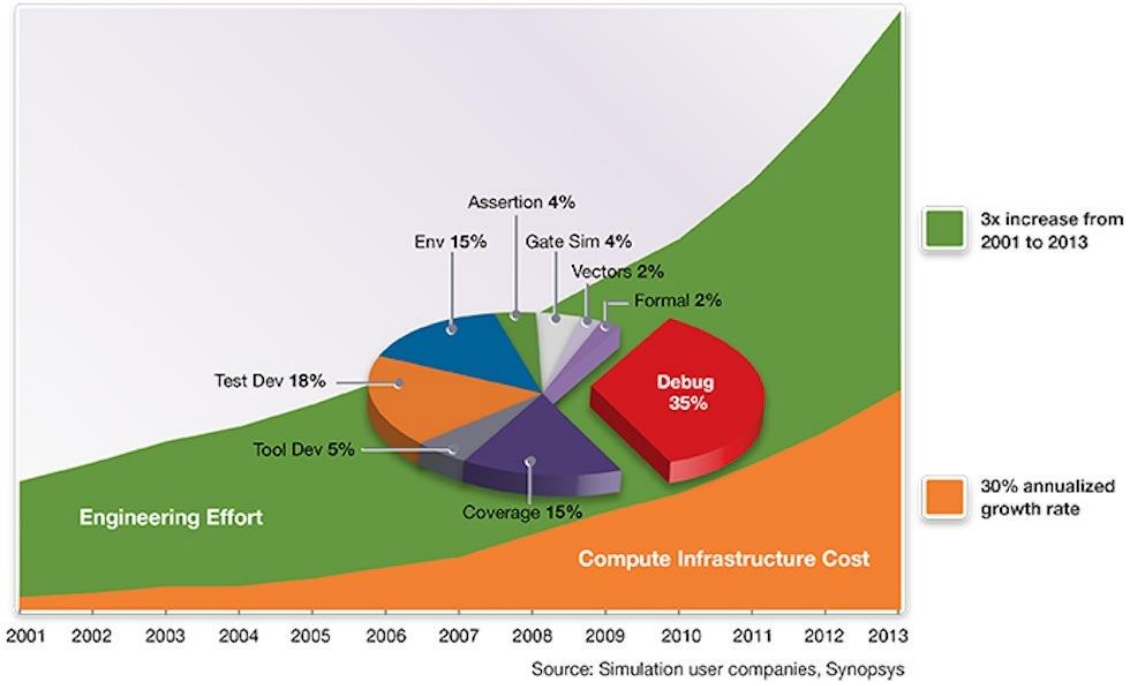


**Figure 2 The Growing Cost of Verification (Source: Synopsys)**

The figure also shows that the engineer effort has increased by three times in the past twelve years. And it is still growing according to SEMICO Research Corp [2].

So, in order to carry out complex designs without missing market windows, a good methodology for design and verification is of great importance to enhance the production efficiency as well as cost saving. This is what some Electronic Design Automation (EDA) vendors, such as Cadence, Mentor Graphics, and Synopsys, have been working on in the recent decades.

### 1.1.1 SoC Design Flow

A typical SoC design flow contains front-end design and back-end design. Front-end design refers to the design activity from system level, or RTL, to net-list level, or gate level. Back-end design refers to the design from net-list level to ready-to-production level. Among each design activity, lots of work on verification is needed to prove the functionality, timing, power, etc. A brief view of typical SoC design is showed in the following figure:

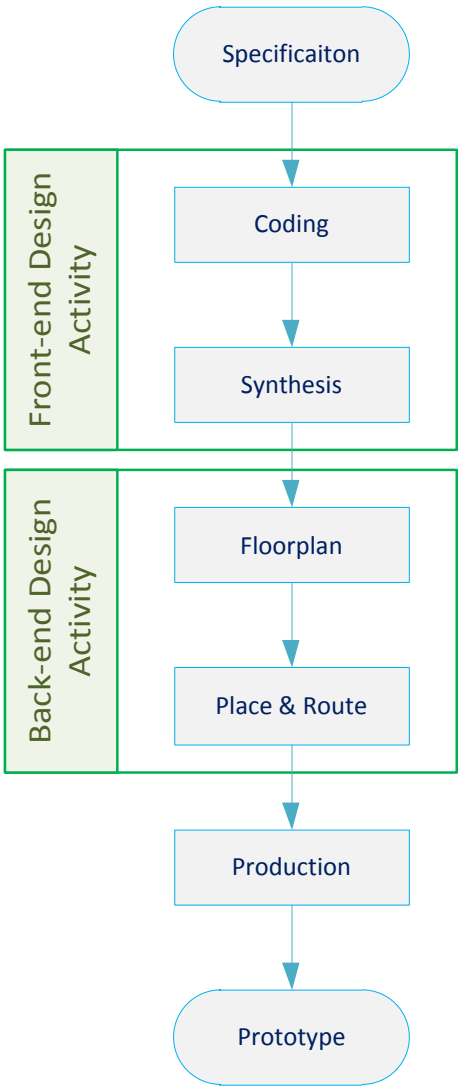


**Figure 3 Typical SoC Design Flow**

Today, big chip vendors are capable to take care of back-end design and provide other design companies with commodity products, such as Digital Signal Processor (DSP), Application Specific Integrated Circuit (ASIC), and Field Programmable Gate Array (FPGA). DSP has the advantage of low cost and ready-into-use. Compared with DSP, ASIC and FPGA are custom made and tailored to the application. ASIC has higher functionality with low power consumption. FPGA has the lower engineering cost of development. Design companies can take use of vendor provided chips and focus on front-end design. All the attention can be paid onto generation of RTL or gate-level net-list.

### 1.1.2 SoC & ASIC Design at Ericsson

Ericsson is one the most successful leading companies in telecommunication area. Ericsson produces most advanced Radio Base Station (RBS) with its own designed WCDMA UPLINK BASEBAND SIGNAL PROCESSING BOARD (RAX) integrated. Several high-tech ASICs and DSPs are embedded on this board for WCDMA (3G) signal processing. Different electronic units are chosen according to the trade-offs, such as performance, power, flexibility, and cost.

In digital design area, Ericsson has adopted the flow which is showed in the following figure:

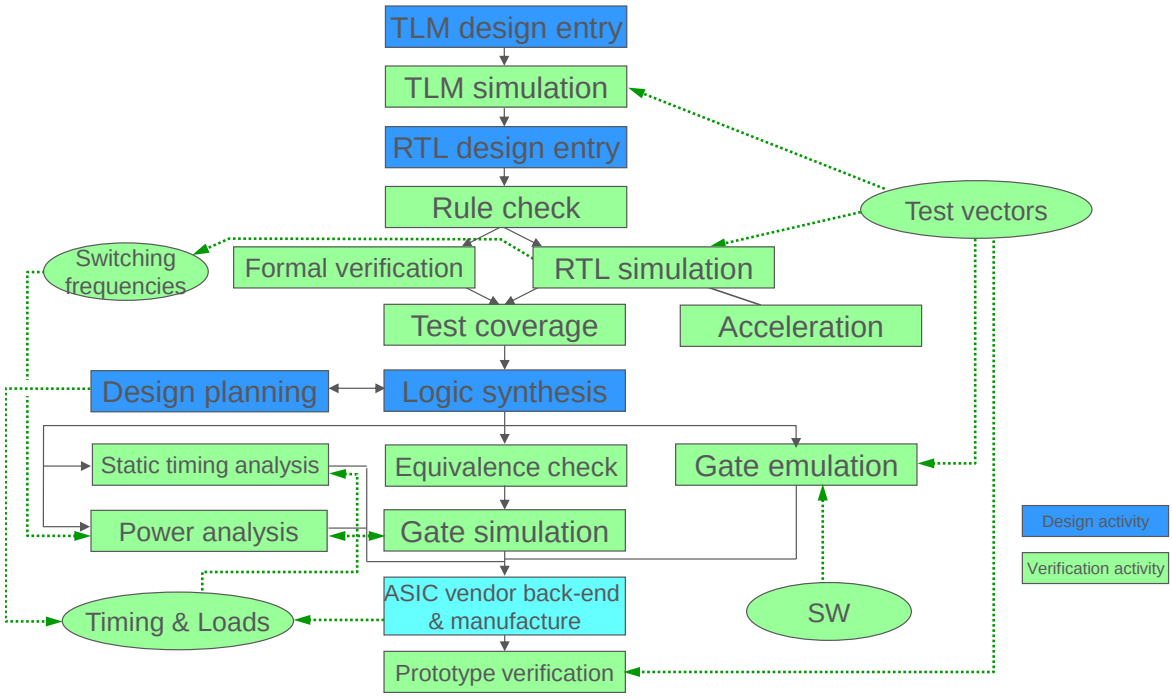


**Figure 4 ASIC Design Flow (Source: Ericsson)**

The flow starts with the TLM design which models functionality. TLM simulation is much faster than RTL simulation (10-1000 times). Then RTL design achieves HW functionality and implementation. Rule check is needed to avoid design patterns that may cause problems in design flows. Formal properties are defined for assertions, constraints, and coverage. Besides formal verification, simulations and debug are carried out based on testbenches to ensure the verification goal.

Once RTL model is ready, logic synthesis can be applied to generate gate-level net-list. The functional equivalence between gate-level net-list and RTL should be checked. Timing analysis, power analysis, and gate-level simulation/emulation are meanwhile handled. Emulation is an accelerated way of verification which is hardware assisted and with optimization. It can be used in RTL and gate-level verification as a complement of simulation.

After all verification goals in gate-level net-list are satisfied, the design will be sent to ASIC vendor for back-end design and manufacture. After the prototype of HW comes out, it still needs to be send back to be verified again.

Verification activities cover most percentage in ASIC design flow and are accompanied with the whole design process. Verification planning starts at the very beginning to identify the features and characteristics of design. Formalities and verification goals are set. Testbenches are written as soon as the plan is done.

Functional verification is the core of whole verification activities. With decades of ASIC developing experience, Ericsson has developed the following functional verification flow:

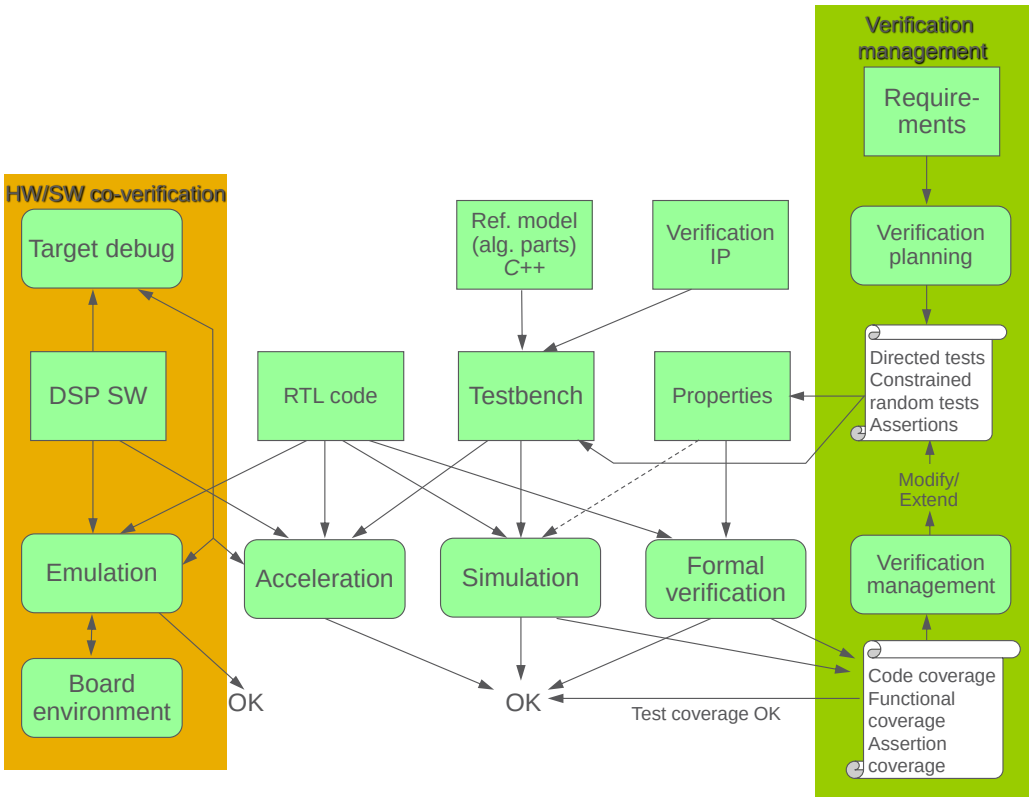


**Figure 5 RTL Functional Verification Flow (Source: Ericsson)**

Mature methodology such as e Reuse Methodology (eRM) and newly established Universal Verification Methodology (UVM) are both adopted. The RTL codes are written in VHDL [3] or SystemVerilog (SV) [4]. Testbenches are written in e [5] or SV. The use of Verification IP (VIP) based on these methodologies in e Verification Component (eVC) and UVM Verification Component (UVC) are used. The property checking uses SystemVerilog Assertion (SVA) and Property Specification Language (PSL) [6]. More of the introduction to these verification technologies will be introduced in Section 2 of this report.

However, this flow is tailored for RTL verification. The development and verification of TLM is parallel to those of RTL. TLM in C/C++/SystemC [7] is mainly used as abstraction and reference model for RTL.

With the development and usage of TLM and HLS, new ways of synthesis procedure is available [8]. Ericsson is looking for the new methodology that will accelerate the design flow from the existing one. Using automated HLS flow to generate RTL from TLM maybe one choice.

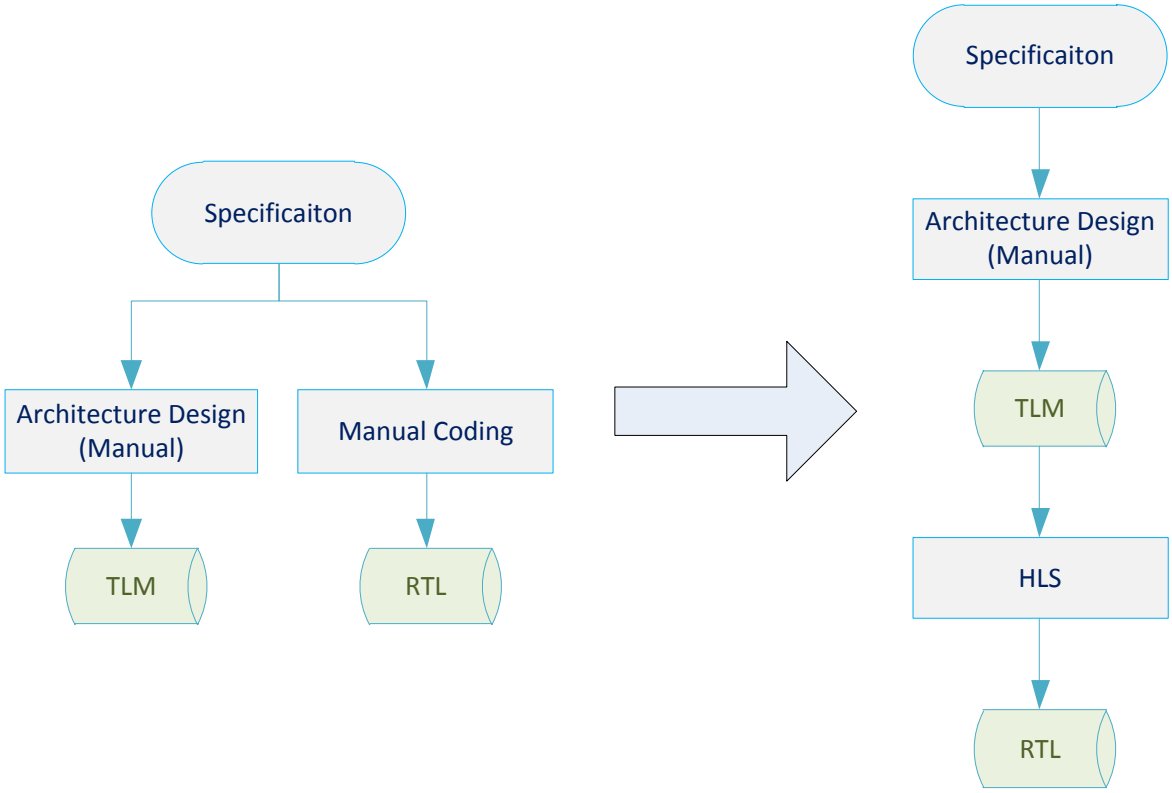


**Figure 6 Using HLS to Automate Design Flow**

This methodology saves the time of writing RTL manually by using the HLS tools provided by EDA vendors. Please refer to [8] for more information.

## 1.2 Problem

The trustfulness of the EDA tool generated RTL is not sure. Although the design automation is realized by HLS tools, the verification process is still needed.

In [8] used some basic simulation techniques to verify the sample RTL. However, more efforts are needed to adapt the RTL code into UVM verification framework.

Since an automated HLS flow is proposed in [8], is there any possibility to adapt automated verification flow into ASIC design needs to be proved.

Can the automated verification flow replace the normal simulation and assertion based verification flow is unknown yet.

And, if there is a feasible way of using automated flow, how much effort it can help to save is to be discovered.

## 1.3 Motivation and Goal

This report aims at devising a flow for verifying the functional equivalence between SystemC/TLM code for ASIC blocks and their corresponding RTL code. The high level models may be in SystemC/TLM/C. The RTL code may be the result of High Level Synthesis. Several verification methodologies have been investigated.

The use of commercial tools that based on sequential equivalence checking has been evaluated and its limitations have been judged. Ways to overcome those limitations by complementary verification methods have be explored and recognized. Tool evaluations are based on the following criteria: verification completeness, adaptability to SystemC and RTL coding style, and ease of use.

# 2 Verification Method

The focus of SoC design contains the following points:

- Operations to perform
- Data to process
- Protocols to follow

In order to ensure these design features, several verification goals are specified at the beginning of design. These goals are turned into different coverage cases in normal verification methods, which usually contain:

- Cover certain percentage of typical cases
- Cover certain percentage of corner cases
- Cover certain percentage of interactions between features

Several verification methods are used to achieve these goals:

- Coverage driven verification: check the overall performance of design based on the percentage of coverage among the specified test cases.
- Constrained random stimulus: generate random cases to run the simulation in a large scale to cover the expected cases.
- Assertion Based Verification (ABV): check the specific behaviors or properties of design. ABV is a way to ensure that the design does not violate certain design rules.
- Hardware-assisted simulation acceleration and emulation: a way to accelerate verification process by using optimized emulation in HW.

In this chapter, a modern verification methodology used in industrial area will be introduced. Some new methodologies or approaches in research area related to TLM-RTL verification will also be discussed.

## 2.1 Universal Verification Methodology

A common verification standard, Universal Verification Methodology (UVM) [9], was brought up in 2010. UVM aims at the reuse of verification components and Verification IP (VIP) for electronic industry. This gives a standardized way for modern system design verification. UVM is basically a class library implemented in SystemVerilog. It is supported by all the leading EDA tool vendors such as Cadence, Mentor Graphics, and Synopsys.

UVM is the combination and unity of Open Verification Methodology (OVM) and Verification Methodology Manual (VMM) for SV. UVM is based on more than ten years' development of verification methodologies. A brief history of the evolution of these methodologies is showed in the figure below:

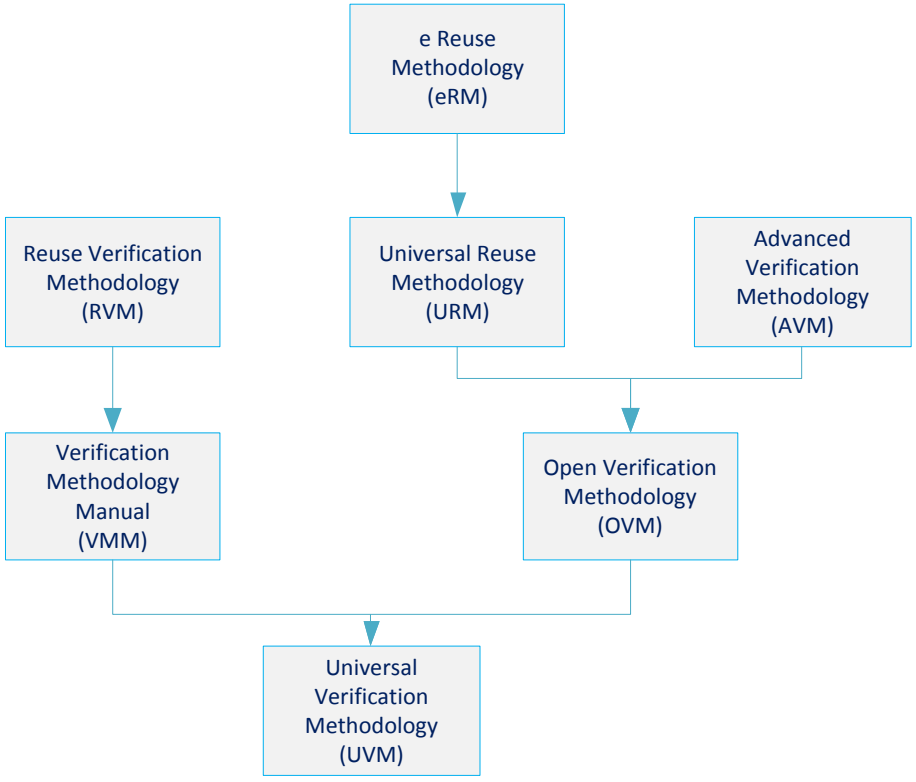
e Reuse Methodology (eRM)
Reuse Verification Methodology (RVM)
Universal Reuse Methodology (URM)
Advanced Verification Methodology (AVM)
Verification Methodology Manual (VMM)
Open Verification Methodology (OVM)
Universal Verification Methodology (UVM)

**Figure 7 History of Verification Methodologies**

UVM has the following advantages:

- The first methodology supported by all three big EDA vendors: Cadence, Mentor Graphics and Synopsys. It will be supported by most of other vendors.

- Standardization on how to build verification environment, create testbenches, generate tests, and collect coverage.
- Best practices which include the reuse of verification components.
- Class Library including SV class component, and IP embedded.
- Large and growing user base
- Open standard and vendor independent

UVM is based on coverage driven verification. It uses reusable and configurable UVM Verification Components (UVC). A basic overview of UVM hierarchy is showed below:

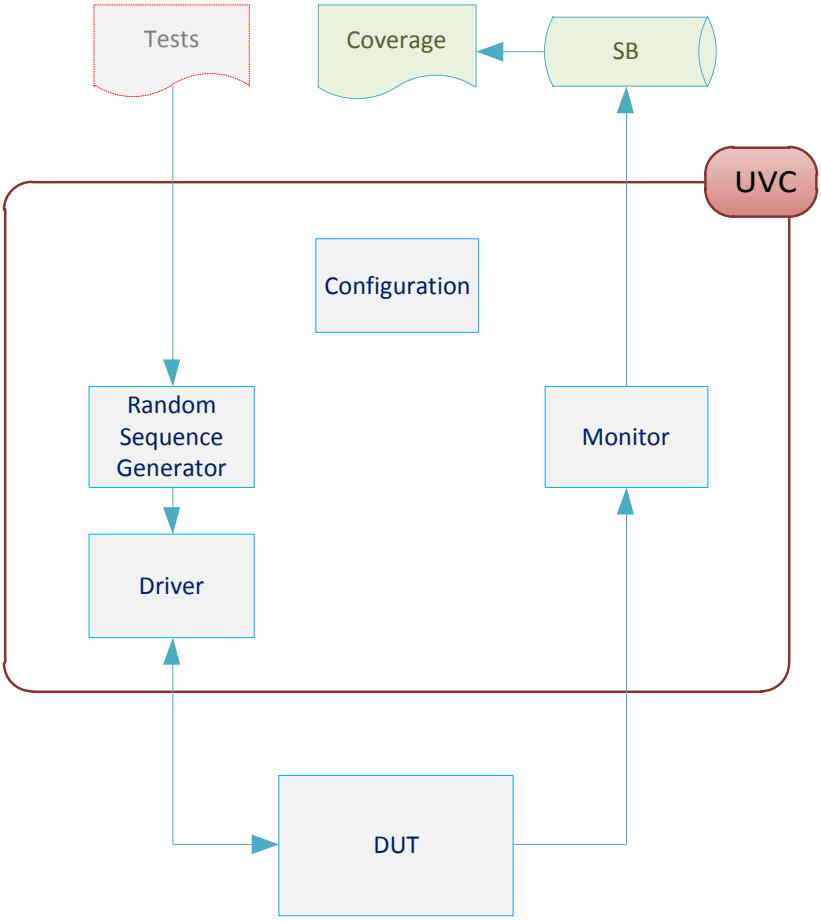


**Figure 8 UVM Hierarchy**

The random sequence generator coordinates the traffic of transaction and generate stimulus from tests for driver. The driver pushes the transaction to the device under test (DUT). The Monitor collects the output from DUT and sends the transaction data to Scoreboard (SB) for coverage checking.

The UVC which contains all the sequence generator, driver, and monitor is configured as active mode. Sometimes, a UVC is configured as passive mode, which only has monitor.

UVM is easily adapted to TLM. All the traffic transactions are transferred in the interface between these components. When UVM is used for RTL models, different UVC is used with additional interfaces or relative components that do the conversion between transactions and signals.

UVM is widely used because of its strong, proven industry foundations, and its capability to verify both RTL and TLM models. However, UVM is simulation based and need a lot of effort on writing testbenches and running simulations. The reuse of TLM testbenches to RTL is not assured. The support for SystemC TLM is not fully featured. A lot of researches have been down and are still being carried out upon the methods for TLM/TLM-RTL verification.

## 2.2 Methods Still Under Research

In this part, some other TLM/TLM-RTL verification methods are discussed. The methods both introduced in 2.2.1and 2.2.2 are the most recent outcomes in research stage. The method described in 2.2.3 is mature and has been realized as commercial tools in industry.

### 2.2.1 TLM Verification

In [10] and [11], a method based on Bounded Model Checking (BMC) for Untimed TLM property check is introduced. It is a possible way for formal check upon SystemC TLM. It also gives a formal transaction-based coverage notion. An overall flow is showed below:

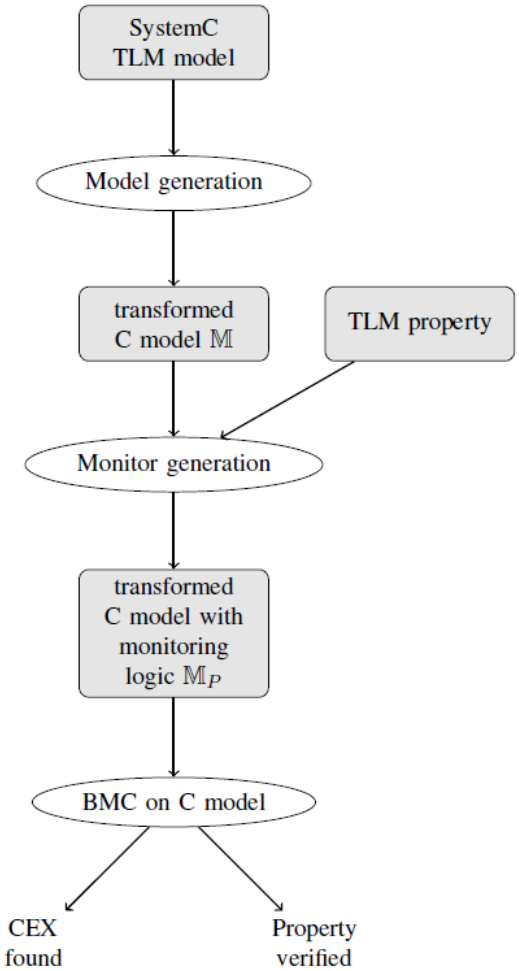


**Figure 9 BMC Based Untimed TLM Property Check (Source: [10])**

However, the SystemC TLM 2.0 construct is not supported. The data coverage is not included either. It cannot ensure the whole functionality of TLM Design.

The same authors also introduced another method in [12] which adapted to the modern verification methodologies, such as OVM and UVM. They introduced their own library: System Verification Methodology (SVM), which made the testbench simulation available for functional verification on SystemC TLM. An example of SVM structure is showed below:

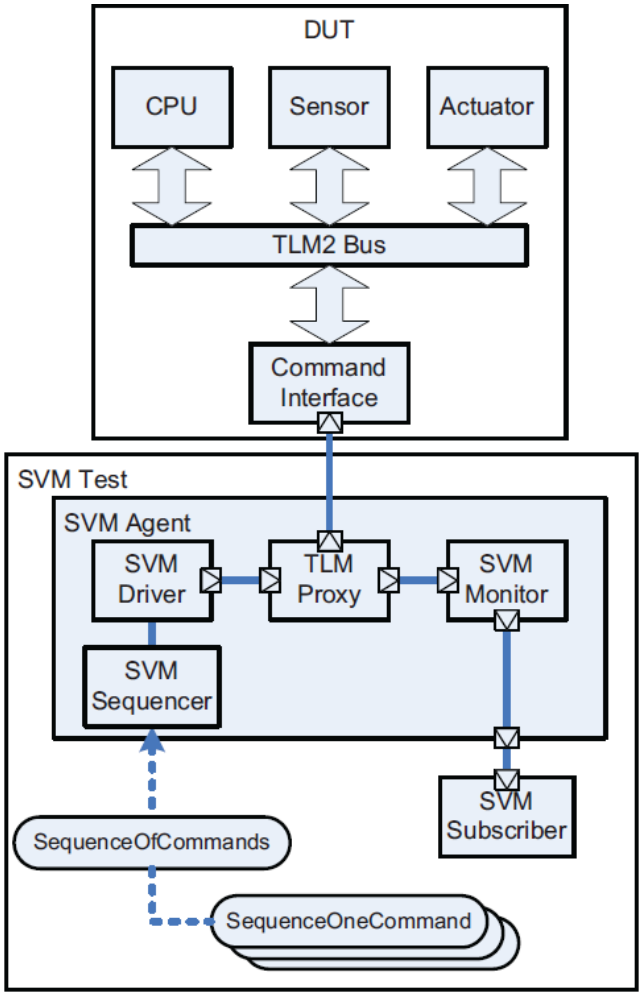


**Figure 10 SVM Structure (Source: [12])**

These two methods propose a possibility for a sound functional verification on TLM design.

### 2.2.2 Assertion-based TLM to RTL Verification Method with Reuse Tool

In [13], a framework for automatic TLM to RTL functional consistency checking is built. This method reuses the TLM ABV and does the automatic refinement to TLM ABV and applies it on RTL verification. The overview of this method flow is showed below:

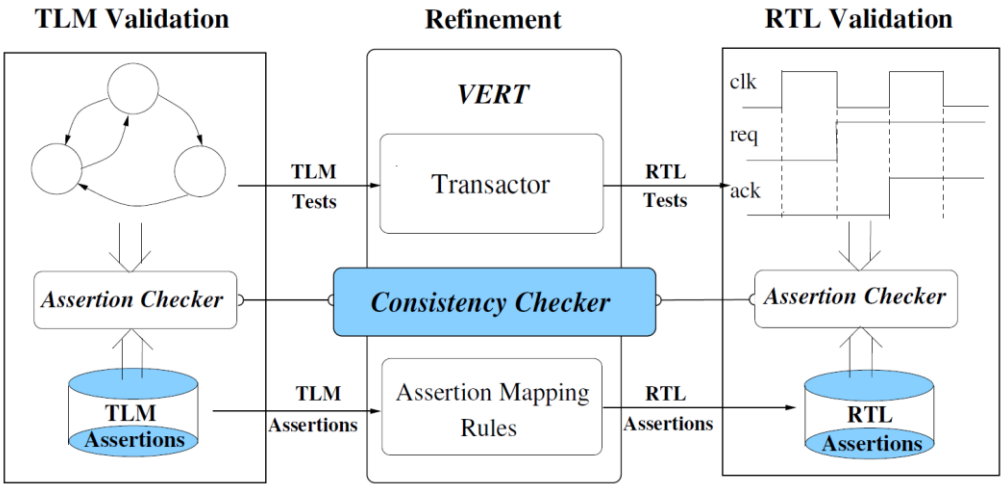


**Figure 11 VERT Consistency Checking Framework (Source: [13])**

VERT stands for *Validation Effort Reuse Tool*. The language is SystemC for TLM, Verilog for RTL and PSL for assertions.

This framework first generates the TLM assertions. Then refinement on assertions from TLM to RTL is performed. Finally, it does the consistency checking by triggering both TLM and RTL assertions together. This framework monitors the assertion coverage as well as assertion ordering to ensure the assertion activity is reliable.

This method has the great potential to reduce the verification and validation effort because the TLM assertions are directly reused for RTL.

### 2.2.3 Verification Tools without Testbench

In [14], the introduction of HLS into design flow and a comparison tool for equivalence checking between design specification in C/C++/SystemC and RTL implementation is mentioned.

With the adoption of HLS automation, RTL can be machine-generated from C/C++/SystemC models. Certain verification tool based on sequential equivalence checking (SEC) [15] can be adapted to the comparison work.

The main reason of using this kind of tool is that it does not require testbench. Compared with RTL simulation, it can perform thorough verification within much less time and effort. It is of great benefit if this tool can be adopted in real design flow. However, the reliability of the tool is yet unknown. To what extent this tool can replace the normal simulation is to be explored.

In the following chapters, this report will focus on the introduction and discussion regarded to this kind of verification tool. Detail demos of using one of the tools will be performed. Evaluation on this tool will be carried out.

## 2.3 TLM-RTL Verification Tools without Testbench

There are several commercial vendors who provide with advanced verification tools. This kind of tools is capable to do verification without normal testbenches simulations, assertions, or coverage. This is an alternative method to do verification in a more efficient and more productive way.

In this section, two commercial tools, HECTOR and SLEC namely, will be introduced. These tools are both based on the SEC.

### 2.3.1 HECTOR

High-level Equivalence C TO RTL (HECTOR) is brought up by Synopsys. It is a formal block-level consistency checker which does not require testbench, assertion, or coverage [16]. This eliminates the normal simulation in verification process and saves the great effort and time spent on it. HECTOR supports blocks written in VHDL, Verilog, SystemVerilog, C, C++, and limited SystemC. SystemC is not fully supported for constructs such as Thread/Method.

HECTOR focuses on data-path but not control logic of the blocks. It is capable of checking following designs:

- Algorithm-intensive blocks
- Designs with float point and bit-serial division
- Data-path dominated designs

HECTOR generates Counter Example (CEX) once there is a bug found. The CEX can dumped as VCD, or can be compiled and run in VCS, which is a RTL language simulation program brought up by Synopsys. HECTOR also provides a debugger (HDB) for C/C++ codes.

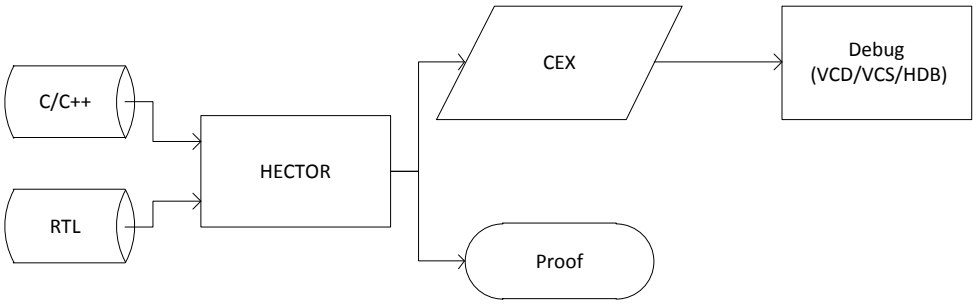


**Figure 12 HECTOR Verification Flow**

More of the description of HECTOR can be found in [16]. However, HECTOR has not been evaluated as part of this thesis work.

### 2.3.2 SLEC

Sequential Logic Equivalence Checker (SLEC) is a functional verification product of Calypto [17] [18]. It aims at functional verification without testbench simulation, which is the same characteristic as HECTOR. Calypto has been developing a verification method between System-level Models (SLM) and RTL [19]. SLEC supports C/C++/SystemC for SLM and VHDL/Verilog/SystemVerilog for RTL.

SLM refers to C/C++/SystemC model that is used as prototypes in SoC development [20]. SLM is usually non-cycle accurate. They behave as different functional blocks in a system, where transactions are done between these blocks without accurate timing information. The way SLEC does verification is basically compare the input/output mapping between SLM and RTL based on each transaction [21]. SLEC is based on the Non-cycle-accurate SEC method between different designs with different times. It uses a unique way that builds cycle accurate unit machine for SLM and RTL designs, which is different from normal verification approaches [22]. A detailed description of this method is presented in section 3.1.

In section 3 of this report, a brief introduction of this method, the way how SLEC is working, and some examples using SLEC to do the functional verification will be presented.

### 2.3.3 Comparison

The common points for both HECTOR and SLEC are:

- Formal verification based on SEC
- Synthesizable Block level model / System level model
- No simulation, assertion required
- Using mapping between C and RTL
- Support black box model
- Support custom constraints
- Support VHDL/Verilog/SV and C/C++
- Generate CEX waveform for bug
- Support synthesizable SystemC
- Do not support SystemC TLM 1.0/2.0

The common points show that both HECTOR and SLEC are very convenient tools which support all the Property Specification Languages (PSL). They also give CEX in waveform to show the bug for debugging. The last two points reveal that have limitation on the C designs. The designs should be synthesizable. For example, function call malloc is not supported.

SLEC also support verification on other field besides HLS, which can be found out in [23]. In the following section, SLEC with HLS for functional verification will be introduced.

# 3 Verification Using SLEC

In this section, following topics will be presented:

- The way how SLEC does verification
- How to use SLEC
- Some demos
- Results, analysis, and estimations of these demos

SLEC supports the integration with HLS tool, such as Catapult C Synthesis provided by Calypto. SLEC also supports the integration with C-to-Silicon Compiler provided by Cadence [24]. This integration offers the designer to make a standardized way to carry out functional verification flow on the fly with HLS flow.

SLEC provides functional verification without testbenches or test vectors. It is also able to detect bugs and create counter examples (CEX) when differences between two designs are founded. SLEC uses sequential logic analysis and mathematical formal algorithms for verification between two models. It makes sure if all input combinations can result in the same output throughout the lifetime of models [25].

In this section 3, SLEC verification flow is based on the Catapult HLS flow. A detail explanation and the way to use Catapult are described in [8].

## 3.1 Principle

SLEC for HLS is based on none-cycle-accurate SEC [22]. None-cycle-accurate means that the operation time between two functional blocks are different. For example, here is a function of a simple adder:

```
void adder(int &a, int &b, int &c, int &d, int &out){
    int temp = 0;
    temp = a + b + c + d;
    out = temp;
}
```

The model for this adder in parallel behavior will be like:

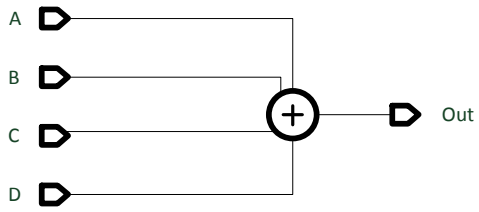


**Figure 13 Simple Parallel Adder**

The time needed to get one output is one cycle.

In another way, the serial model for this adder will be like:

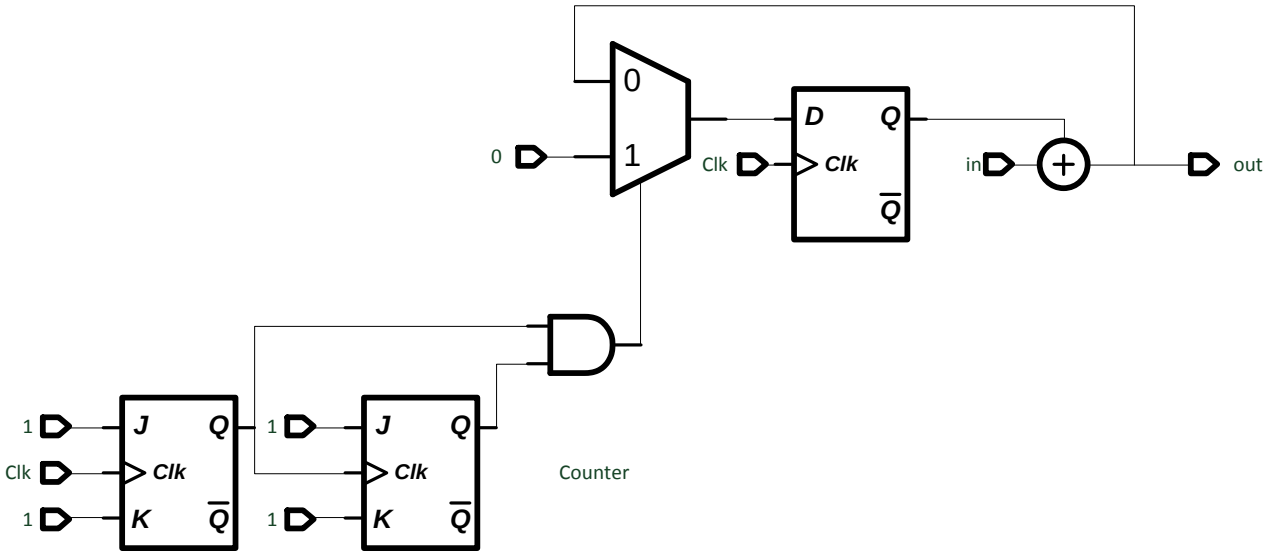


**Figure 14 Simple Serial Adder**

This will take four cycles to get an output. A counter counts from 0 to 3 during every period. The input of the register is reset to 0 every time when counter counts to 3.

The adder, either parallel or serial, can be considered as a transaction, the functionality of both two adders is the same. But the timing is different. This kind of situation is universal in TLM-RTL design. TLM is only I/O-accurate, whereas RTL is pin-accurate. Different level of timing accuracy requires a great effort to do the verification based on normal simulation.

SLEC does none-cycle-accurate SEC by constructing unit product machine and time correlating. This time correlated unit product machine is an abstract internal model built by SLEC.

A detailed algorithm for constructing unit product machine is described in [22]. This unit product machine is basically a refined sequential machine with zero latency on output and its period is also one. It is the most key part which includes all the core information for SEC, such as I/O maps, periods, and state maps. Through the unit product machine, the joint maps are built between two different designs irrespective of their timing difference. For example, the unit product machine for both the simple parallel and serial adder is showed in the following figure:

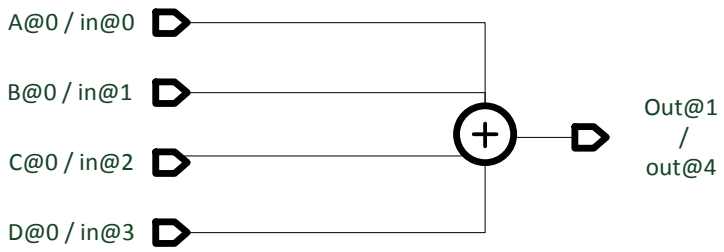


**Figure 15 Unit Product Machine for Simple Adder**

All the inputs and outputs are mapped in this design. The inputs in parallel adder at time 0 are mapped to the inputs in serial adder with corresponding inputs at different timing. The output in parallel adder at time 1 is mapped to the output in serial adder at time 4. Since the flops in serial adder are set to the reset value each period, they are eliminated by the algorithm '*Optimize State Variables*' in [22], thus no other state (flop) map is presented here.

However, state maps usually consist in the unit product machine and it usually requires time correlation during the construction. This is because the design may have non-one period or non-zero latency inputs or outputs. The detailed algorithm for time correlation is skipped in [22].

This method creates cycle accurate unit machine for different models. As long as this unit machine is built with correct output maps, the two different models are proven to be equal. Any unit machine with unmapped output is regarded as a counter example ('*Theorem 1*' [22]).

In normal verification process between SLM and RTL models, both of these models are refined by constructing unit product machine with time correlation upon them. The unified timing control is basically a wrapper for both SLM and RTL models, which is showed in Figure 19. After non-cycle-accurate SEC is carried out, the output maps are compared to get a proof of both designs. The detailed process of using SLEC will be discussed in the next section.

## 3.2 Verification Plan

The aim of automated HLS flow is to simplify the verification flow by greatly removing the test based simulation. A brief concept of the simplification is showed below:

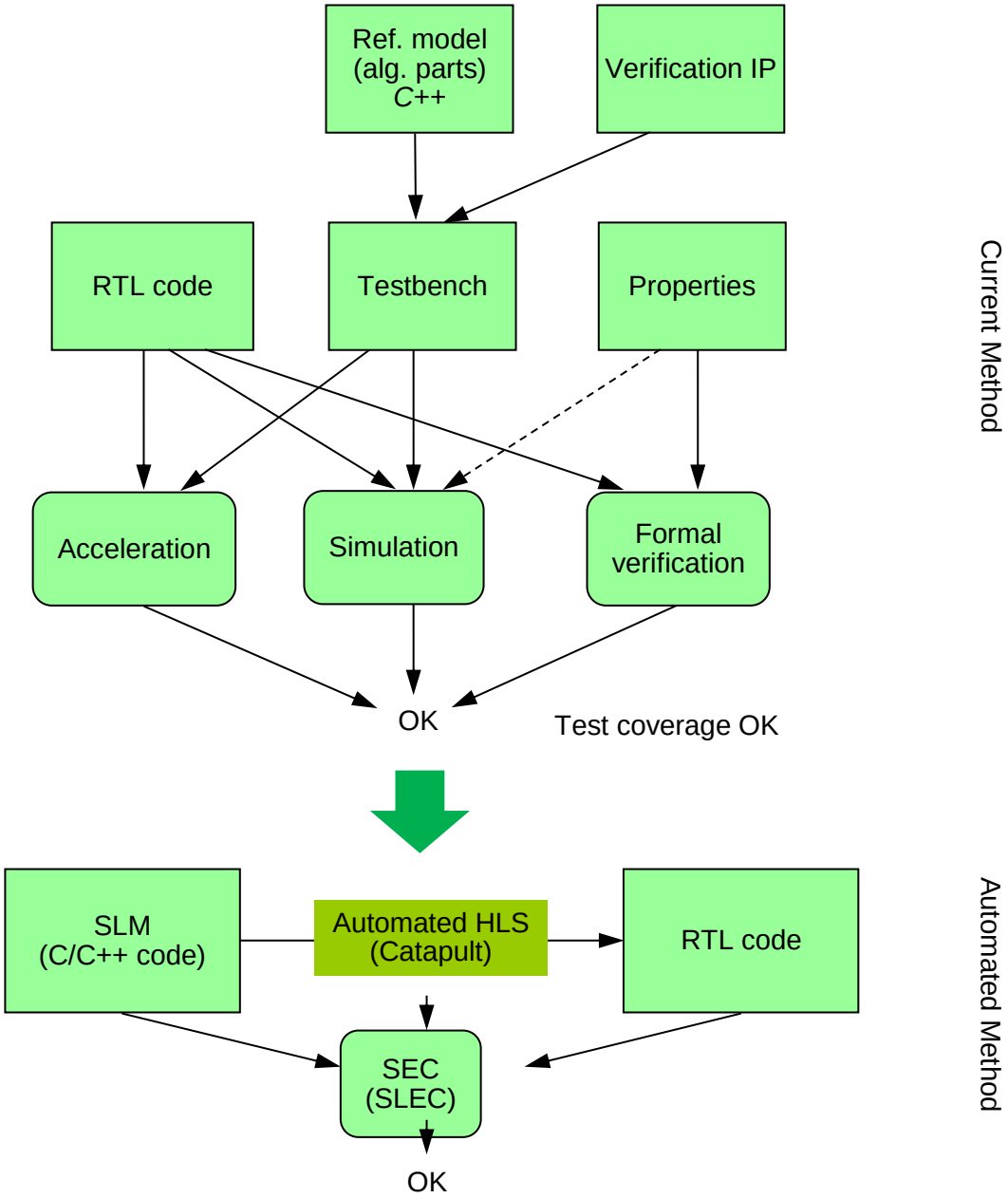


**Figure 16 Simplify the Verification Flow**

As is discussed in [8], SLM can be directly used as input of automated HLS flow and RTL code can be generated by EDA vendor HLS tools, e.g. Catapult, CtoS, etc.

In this report, Catapult is chosen to build up the design and verification flow environment. The reason to choose Catapult is that SLEC script file can be generated by Catapult during HLS. The script file includes all the design constraints that have been used and these are all the reference input to boost the logic process of SLEC. This is because the design constraints are chosen according to different libraries corresponding to different HW structures. The timing and pipeline information are also needed since these are also introduced to generate RTL code. Users can also modify the script file to add more properties in order to have alternative checkpoints.

The whole picture of this project is to see if and to how much extent SLEC can be relied on to perform the SLM to RTL verification without testbenches. And what are the “must know” points when using SLEC.

The following chapters will introduce the brief way of performing equivalence checking using SLEC. Different examples are used during the tests so that the pros and cons of using SLEC can be easily shown.

## 3.3 SLEC Verification Flow

SLEC verification flow consists following basic steps:

- Define log output formats
- Create a Tcl file
- Read the designs
- Apply any required constraints to the designs
- Run verification
- View Reports
- Analyze the output (counterexample waveforms and testbenches)

The above steps now are integrated with Catapult HLS Flow and most generation parts are fully automated. It can always be manually configured for specific goals. The manual flow is stated in SLEC user manual and is not presented here.

The following figure shows the overview of SLEC-Catapult Design Flow:

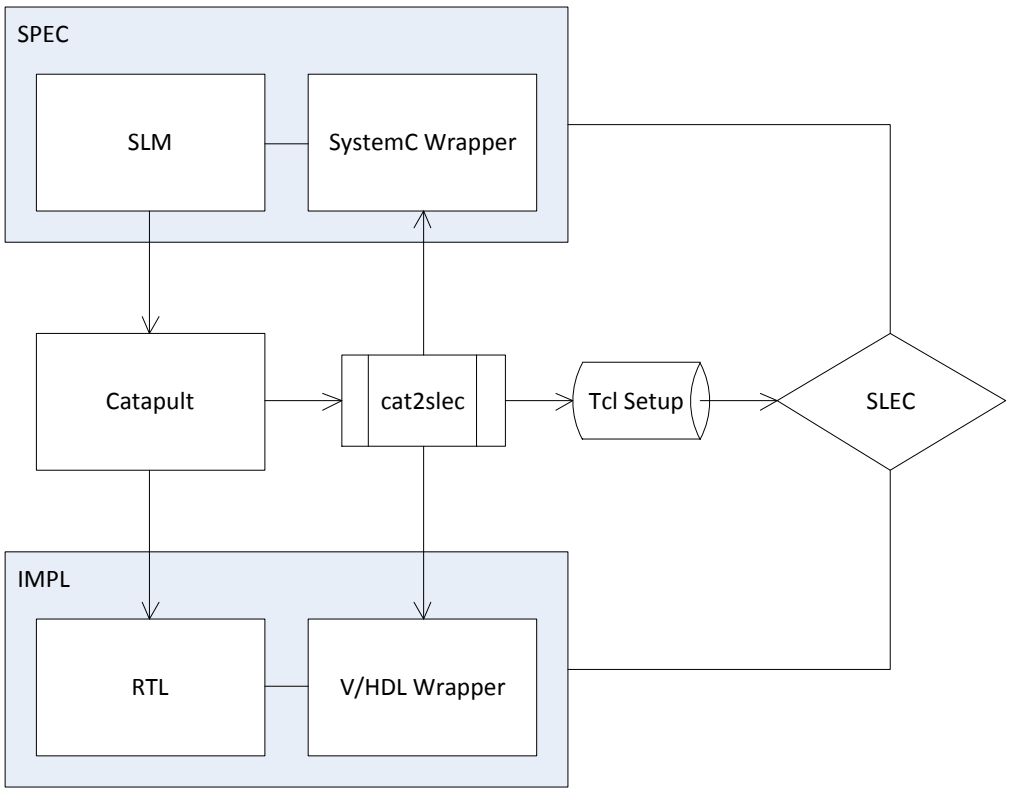


**Figure 17 SLEC-Catapult Design Flow**

The concept of design library is used by SLEC:

| | |
|---|---|
| **SPEC** | the specification design library |
| **IMPL** | the implementation design library |
| **CONS** | additional design constraint modules (optional) |

**Table 1 SLEC Design Library**

SPEC includes the design used as reference model for verification. In HLS flow, SPEC is the input of HLS tool. IMPL includes the design that wanted to be verified. In HLS flow, IMPL is the output of HLS tool. CONS is the specification of additional constrains on design modules. It enables custom constrain upon SPEC or IMPL during verification. CONS is an optional design library.

Following steps are the detail instruction of SLEC verification flow:

- Using Catapult C synthesis to generate RTL (IMPL) from SLM (SPEC). SLM is written in ANSI C/C++ which is synthesizable. RTL can be Verilog/VHDL. For detail description of this step, please refer to [8].

- Enable the SLEC Verification Flow in Catapult. This can be both done in Catapult GUI (Figure 18) or using the following Tcl command:

```
# ##########################################
# Enable the SLEC integration
flow package require /SLEC

# SLEC verification options
# enable full proof mode (default is find_error mode)
flow package option set /SLEC/ENABLE_FULL_PROOF true

# enable flop maps (default is false)
flow package option set /SLEC/ENABLE_DB_FLOW true

# ##########################################
# Required Catapult options for SLEC!!!

# Disable Multiplier Decomposition
directive set -OPT_CONST_MULTS 0

# Disable CSA adder Optimizations
directive set -CSA 0
```

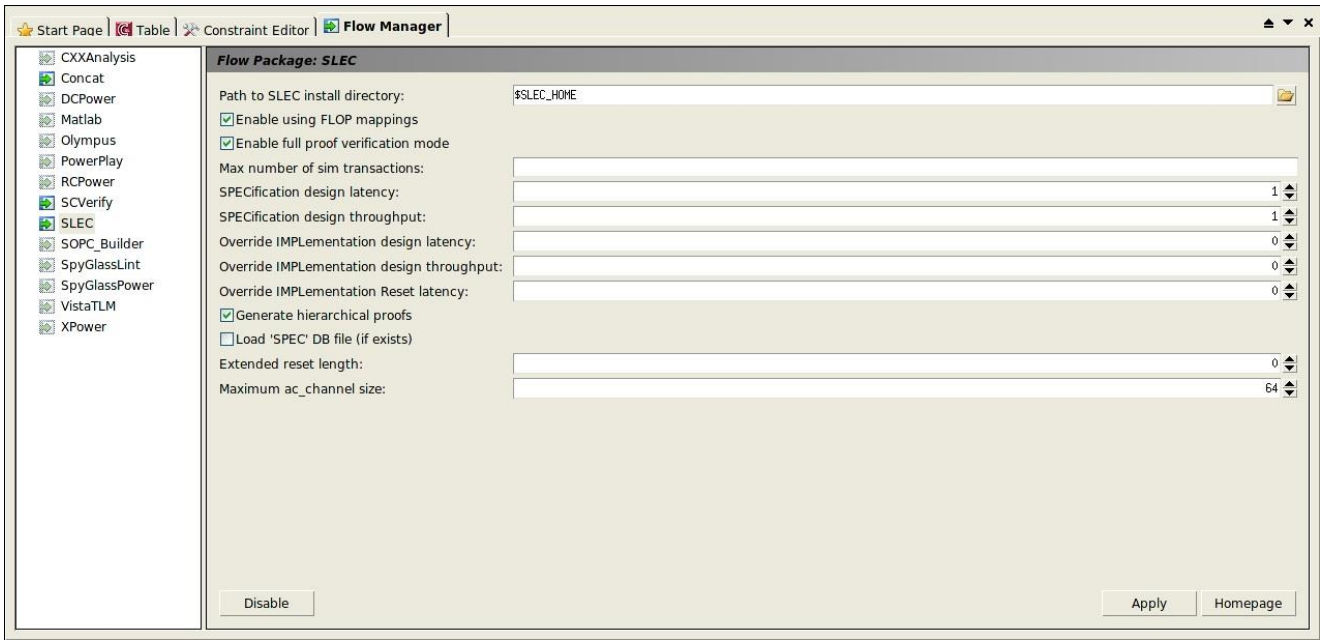


**Figure 18 Enable SLEC Flow in Catapult**

- Run the cat2slec compiler to prebuild the environment for SLEC. It will do the following jobs:

  - Generate wrapper for both SPEC and IMPL. This is to ensure the bit accuracy. Timing information is also added by adding a register for the output of SPEC. Figure 19 shows a simple example of the structure of cat2slec generated wrapper.

    – spec_wrapper.cpp : wrapper for SPEC

    – impl_wrapper.v(hdl) : wrapper for IMPL

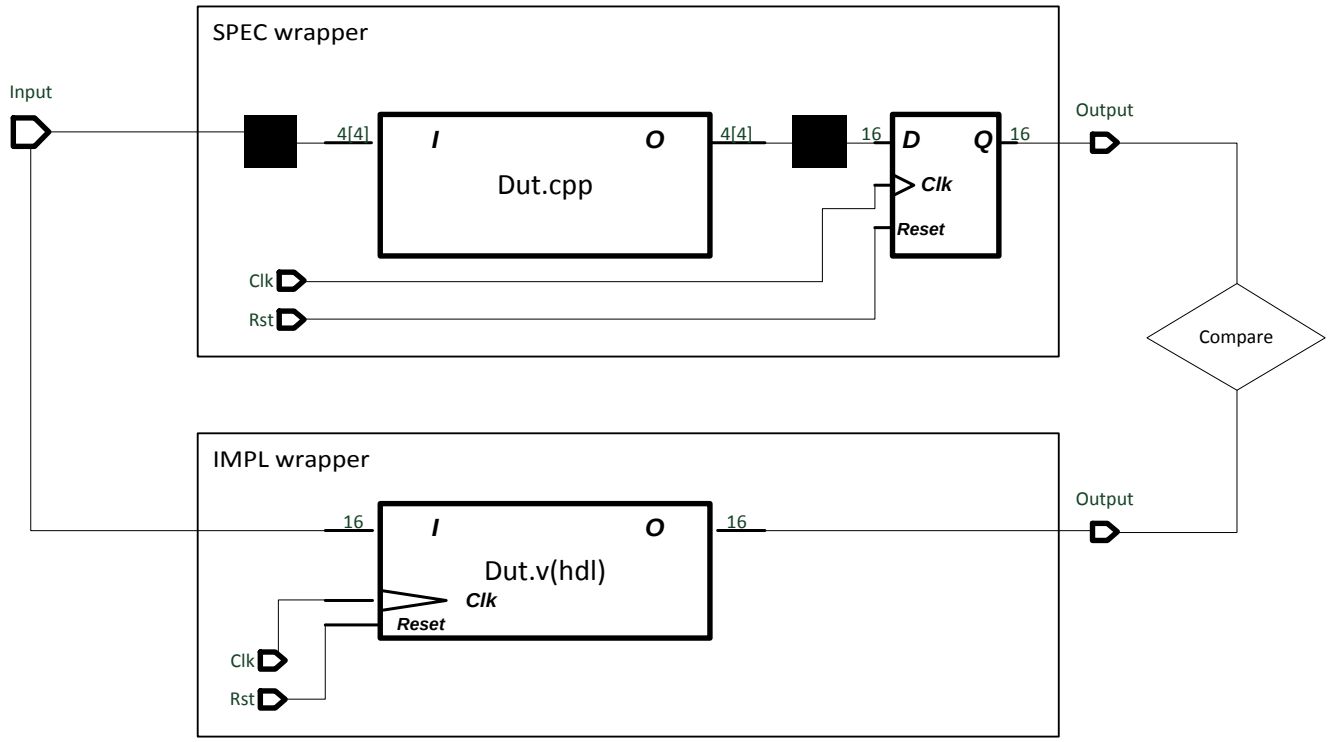


**Figure 19 cat2slec Wrapper Example**

  - Generate the Tcl scripts for C-C and C-RTL verification. These Tcl files are used for controlling SLEC during verification. They can be manually adjusted. An example will be discussed in detail in section 3.4.

    – SLEC_compile_rtl_v(hdl).tcl : used for C-RTL verification.

    – SLEC_compile_c_to_c.tcl : used for C-C verification

- Do refinement on throughput and latency. If the design uses ac_channels, cat2slec also determine the optimal sizes of the ac_channels.

- Apply constraints in CONS to pre-build the verification flow. These constraints files should be in the same folder with source SPEC files. They provide user to specify the control of SLEC. These constraints will be automatically sourced once cat2slec flow is invoked. An example of the usage of these constraints will be discussed in section 3.4.

To invoke the cat2slec compiler flow in the shell under folder <project-folder>/Catapult/<solution-folder>.v<n>/SLEC/:

```
$ ./cat2slec.sh vlog/vhdl [option]
```

cat2slec has switches for customize options:

| Option Name | Description | Notes |
|---|---|---|
| **-help** | Prints the usage options | |
| **-workdir** | Name of the Catapult SOLUTION_DIR directory. | This is a required option to execute the script. |
| **-spec_block** | Specifies the name of the C++ function which has #pragma design" or "#pragma design top" set on them. | This option is required to execute the script. When *–hier* is not provided then the function name specified must be the design top. |
| **-impl_block** | Specifies the name of the Verilog/VHDL module created by the synthesis of the C++ function. | |
| **-rtl_lang** | Specifies the language of the RTL netlist for which the SLEC setup is required. | Valid arguments to this switch are vlog and vhdl. Default: vlog. |

| | | |
|---|---|---|
| **-num_trans** | Specifies the number of SLEC transactions during refinement. | Default: 20 |
| **-timeout** | Use to mention time out value of the SLEC refinement run. | Default: 1800 seconds |
| **-norefine** | Disables refinement runs. | Legal values: all \| tp \| lt \| ac \| flop.<br><br>No Default value. |
| **-disable_flop_maps** | Disables use of flop maps. | By default, use of flop maps is enabled. |

**Table 2 Options to Customize cat2slec Flow**

For example, to invoke the flow without ac_channel re-sizing and wrapper re-generation:

```
$ ./cat2slec.sh vlog -noac -nowrap
```

- Invoke the SLEC verification flow using generated Tcl scripts for C-C and C-RTL verification. Tcl scripts are also customizable. User defined CONS constraints will also be loaded automatically.

  To invoke the cat2slec compiler flow in the shell under folder <project-folder>/Catapult/<solution-folder>.v<n>/SLEC/:

  For C-C:

```
$ slec SLEC_compile_c_to_c.tcl
```

  For C-RTL:

```
$ slec SLEC_compile_rtl_v(hdl).tcl
```

  After SLEC verification flow is finished, several output log will be generated in <project-folder>/Catapult/<solution-folder>.v<version number>/SLEC/Calypto/ folder by default. There is an example in Appendix 0.

If SLEC find out any falsified pair of output maps, it will generate Counter Examples (CEX) with VCD files, testbenches, and Makefile automatically. You can run the Makefile under the Calypto/ folder and see the waveform using simulation tools.

This provides you with an easy and efficient way to debug. The following table shows the basic file structure for SLEC verification flow:

| Folder Name | Description | Files Contained |
|---|---|---|
| **<project-folder>/** | Design libraries are located here, including SPEC and CONS. | Catapult Tcl file.<br>Source ANSI C/C++ files.<br>Custom constraints Tcl files. |
| **Catapult/** | Catapult project location | |
| **<solution-folder>.v<n>/** | Catapult output IMPL design files. | rtl.v (IMPL) |
| **SLEC/** | SLEC configure and running directory | cat2slec.sh<br>spec_wrapper.cpp<br>impl_wrapper.v(hdl)<br>SLEC_compile_c_to_c.tcl<br>SLEC_compile_rtl_v(hdl).tcl |
| **Calypto/** | SLEC output directory | slec.log<br>CEX (VCD, testbenches) |

**Table 3 File Structure of SLEC Verification Flow Environment**

## 3.4 Verification Examples

In [22], a few small industry samples, such as an FIR filter and some simple video decoders had been tested and verified using SLEC. However, these are far from enough to say that SLEC can make successful verification on real industrial designs.

The facilitation of SEC also requires that the TLM/SLM should be statically abstracted. A coding guideline had been proposed in [19] [21]:

- Use standard libraries
- Avoid using dynamic memory allocation
- Avoid using pointer aliasing
- Constrain variable loop bounds

In HLS flow [8], the former two problems have been resolved. The introducing of Bit-Accurate Data Types makes the C++ model behave as a true HW [26].

However, pointer aliasing and variable loop bound has not been verified yet. How much these will affect the verification flow of SLEC is to be explored.

SLEC verification procedure will be carried out to verify between the SLM C code as the input before HLS, and RTL as the output after HLS. They are supposed to be equal and can be verified successfully by SLEC because the HLS flow is purely automatic and carried out by mature vendor tools.

In the following two examples, the test and estimation of SLEC are carried out. Assessment of usability of SLEC is drawn.

### 3.4.1 Ericsson DERM Block

De_Rate Matching (DERM) is a sub-block inside an Ericsson design. In [8], a synthesizable SLM is derived from another TLM 2.0 SW model according to this algorithm. Three problems were resolved:

- Avoid using function call malloc
- Create bit-accurate interfaces
- Optimize the code for required HW resources

This SLM is written in ANSI C++ in order to adapt to the HLS tool.

#### 3.4.1.1 Code Structure

The outlook of main functionality is presented below:

```
/*......*/
void DERM(int nrof_rows, /*other column permutation control
interfaces*/) {
    for (/*......*/) {  //column control, static number of columns
        loop_write_s : do { // write systematic bits into columns
            // get systematic bits (s) from input stream in the size of
8 bytes
            // insert dummy and filler bits
            // compute de-interleaved address
            // store bits to de-interleaved address
            row ++;
        } while (row < FUNCTION(nrof_rows))
    }
    for (/*......*/) {  //column control, static number of columns
        loop_write_p : do { // write parity bits into columns, both p1
and p2
            // get interlaced parity bits (p1, p2) from input stream in
the size of 8 bytes
            // insert dummy and filler bits
            // compute de-interleaved address
            // store bits to de-interleaved address
            row ++;
        } while (row < FUNCTION(nrof_rows))
```

```
    }
    for (n = 0; n < 3; n++) { // do these steps three time for s, p1
and p2
        loop_read : do { // write parity bits into columns, both p1 and
p2
            for (/*......*/) {  //column control, static number of
columns
                // read bits in linear order from memory
                // remove dummy bits
                // put bits to output stream in the size of 8 bytes
            }
            row ++;
        } while (row < nrof_rows)
    }
}
/*......*/
```

There are three main do-while loops which use variable loop bound:

- loop_write_s : write systematic bits to local memory matrix
- loop_write_p : write parity bits to local memory matrix
- loop_read : read from local memory matrix to output

Special attention should be paid to these loops. Loop maximum iteration number should be evaluated according to the HW resources before setting up SLEC environment.

SLEC enables customize constraints on loop bound operation such as defining unroll_loop_with_limit. This operation specifies the maximum number of times that a loop will be unrolled. Then it can be calculated to generate output map with fixed number of iterations. This is a way to deal with variable loop bound. It will be verified through the following examples.

#### 3.4.1.2 Verification using SLEC

First, the RTL is generated from Catapult using the SLM C code. From section 3.4.1.1, it can be clearly seen that three do-while loop are not statically bounded. These three loops are labeled as loop_write_s, loop_write_p, and loop_read separately. According to the testbench used in [8], the NrOf_Row loop bound variable is 2. To make it safe, the maximum number of times these three loops are unrolled was set to 3. The NrOf_Row variable was also constrained to 2 during verification process.

As a reference, a detailed way to define constraints files for CONS will be discussed in section 3.4.2.2.

After Invoking the SLEC verification flow using generated Tcl scripts C-RTL verification (SLEC_compile_rtl_v.tcl, section 0), there are still some problems popped up during SLEC running:

- Turning on refinement when running cat2slec will cause infinite running and abortion of SLEC building process. In the log file, infinite out of bound access (OOBA) errors were found.
- Although refinement was turned off and SLEC verification environment was built successfully, the C-RTL verification process still fell into infinite calculation and aborted. Also in the log file, infinite OOBA errors were found.

In order to make sure that the SLM does not have combinatorial logic error or channel access error, a C-C verification against itself is needed. The C code is defined both as SPEC and IMPL. Since SLEC do checks based on I/O maps and State/Flop maps between SPEC and IMPL, it can tell if the SLM satisfy the properties that need to be checked. If there is no error, then the SLM is suitable for further C-RTL SEC.

In Tcl script (SLEC_compile_c_to_c.tcl), property_checks was turned on. Out of bound access (OOBA) check on array accesses was also added before calling verify.

```
# Put the following command under # setup label
set_verification_mode -catapult_flow -property_checks

# Put the following command before calling verify
check_properties -spec -prop -oob_access
```

After running C-C verification, a lot of OOBA failure and the process aborted automatically after about one hour. SLEC does not provide ways to debug OOBA failure. Simulation and manual assertion is needed.

This cannot assure that the design has faults. The reason of OOBA failures may be caused by the pointer alias.

In DERM code, a generic array/pointer for storing the data addresses is used. These temporary pointers are used both in loop_write_s and loop_write_p:

```
unsigned int c_addr[8] = {0};
```

In output rtl.v of the Catapult HLS, these pointers are synthesized to registers:

```
reg [6:0] loop_write_s_c_addr_7_sg4_sva_9;
reg [1:0] loop_write_s_c_addr_7_1_sva_9;
reg [6:0] loop_write_s_c_addr_6_sg4_sva_9;
reg [1:0] loop_write_s_c_addr_6_1_sva_9;
reg [6:0] loop_write_s_c_addr_1_sg4_sva_9;
reg [1:0] loop_write_s_c_addr_1_1_sva_9;
reg [6:0] loop_write_s_c_addr_5_sg4_sva_9;
reg [1:0] loop_write_s_c_addr_5_1_sva_9;
reg [6:0] loop_write_s_c_addr_2_sg4_sva_9;
reg [1:0] loop_write_s_c_addr_2_1_sva_9;
reg [6:0] loop_write_s_c_addr_4_sg4_sva_9;
reg [1:0] loop_write_s_c_addr_4_1_sva_9;
reg [6:0] loop_write_s_c_addr_3_sg4_sva_9;
reg [1:0] loop_write_s_c_addr_3_1_sva_9;
```

The above code shows all the registers generated for pointers c_addr. From the naming method, it is clear that these registers are generated for the memory mapping pointers during the HLS of loop_write_s.

However, in loop_write_p, RTL does not generate new registers for the memory mapping pointers. MUX is used instead for redirecting the pointers in loop_write_p to the registers generated during loop_write_s. Here is one example which shows the redirecting of one pointer in loop_write_p to the one in loop_write_s:

```
assign mux1h_462_nl = MUX1HOT_v_7_5_2({(reg_147[6:0]) , reg_307 ,
loop_write_p_for_3_mux1h_94_itm_sg2
      , (z_out_40[9:3]) , loop_write_s_c_addr_7_sg4_sva_9},
{(or_4380_tmp & (fsm_output[1])
```

```
      & nor_1759_m1c) , ((fsm_output[2]) & nor_1759_m1c) ,
((fsm_output[3]) & nor_1759_m1c)
      , ((fsm_output[4]) & nor_1759_m1c) , and_dcpl_2304});
```

In the highlight parts, the map of port to the generated register for pointer c_add[7] is used in this assignment. This port is a part of MUX, which shows the reuse of registers. This kind of structure will cause the HW behavior which is pointer alias. This will affect the combinatorial calculation of SLEC on the memory access and cause the OOBA errors.

SLEC does not supply ways to control to logic of pointer. Simulations and manual assertions on range test are needed in order to verify the memory access.

#### 3.4.1.3 Summary

Due to the abortion of calculation, SLEC cannot be used for further verification on DERM. The problem of pointer alias can cause SLEC to crash down. One way to settle the pointer aliasing problem is that to create independent pointer for each memory access. However, this will increase the number of registers in RTL, thus increasing the area of the design. This will affect the leverage of design. It is better to use simulation and assertions for verification in this case.

The problem of variable loop bound also remains unresolved. It is further discussed in the next section.

From DERM example, limitations had been found to adapt SLEC to the design which does not follow the guidelines of hardware intent SLM/TLM.

The following condition will cause the failure of SLEC verification:

- Not using standard libraries

  So far SLEC cannot fully support the TLM 2.0 model. The design flow is highly dependent on the standard EDA libraries.

- Using dynamic memory allocation

  Functions such as malloc() cannot be used in the SLM model which doesn’t have a specific HW scaling value.

- Using pointer aliasing

  Generic pointers will cause problems for SLEC verification since same registers are used in RTL for different purpose and this introduce the confusion of logic checking.

- Unconstrained variable loop bounds

  While loop will cause the calculation turns into endless circulation.

- Channels are not bit-accurate

  Interfaces should always be standard. Channels that are not consistent in size will also cause the failure.

If the above attentions have been taken care of, is SLEC trustable to do the verification? Here comes the second example in the next chapter to prove the hypothesis.

### 3.4.2 Simple Transaction Block with Memory via Channel

This is an example especially designed for testing the variable loop bound. It is using the standard channels and with proper loop limits control. There is no generic pointer and no dynamic memory allocation. The block structure is showed as below:

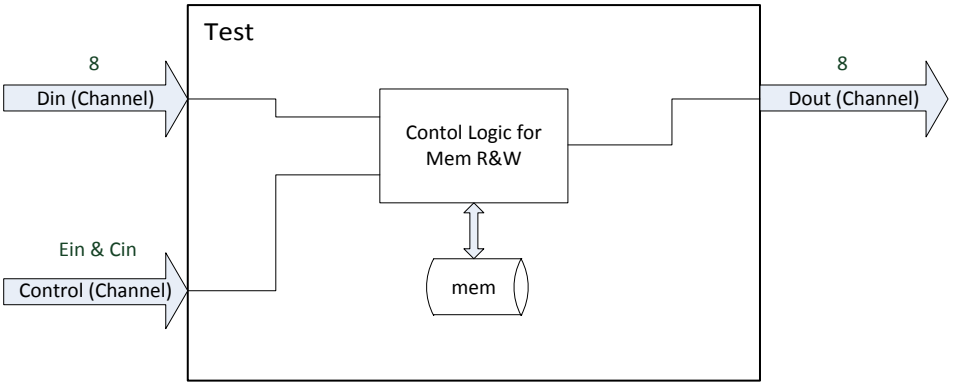


**Figure 20 Simple Transaction Block**

This block is simply a 'read in order then write in controlled' machine. It stores the read data in an array which can be synthesized as an external memory. The output is then got from this array in a new order which is controlled by Ein and Cin variables. This module is designed in order to test the memory access and variable loop bound.

In the module, the variable Ein mainly controls and switches the read or write pattern for array operation. The variable Cin controls the loop bound, which define the data quantity of the array during reading and writing. All the channel interfaces and control parameters are with fixed bit width. The size of memory is also fixed.

#### 3.4.2.1 Code Structure

For a complete code of this design, please refer to Appendix 7.1.1.

```
//preliminary declaration

struct Control {
    ac_int<3, false> ein; //
    ac_int<8, false> cin; //
};

#pragma hls_design top //for Catapult HLS
void test(  ac_channel<ac_int<8> > &din, //channel for input data
```

```
            ac_channel<Control > &c_in, //channel for input control
parameter
            ac_channel<ac_int<8> > &dout //channel for output data
            ) {

    // Initialize the memory array with 128 bit-width;

    if (c_in.available(1) ) { // check input channel validity

        //internal variable declaration;

        if ( c_in.ein > 0) {

            count_loop1 : for(/*...*/){/*...*/} //read c_in.cin number
of data from din and write the data to the memory;

            if ( c_in.ein == 2) {
                count_loop2 : for(/*...*/){/*...*/} //read c_in.cin
number of data from the memory in sequential order and write the data
to dout;
            } else {
                count_loop3 : for(/*...*/){/*...*/} //read c_in.cin
number of data from the memory in reverse order and write the data to
dout;
                count_loop4 : for(/*...*/){/*...*/} //read c_in.cin
number of data again from the memory in sequential order and write the
data to dout;
            }
        }
    }
}
```

There are four loops using variable loop bound:

- count_loop1 : cin number of data are read from channel din and then stored into local memory array.
- count_loop2, count_loop3 : cin number of stored data are read from local memory and then write to the output channel in a sequential order.

- count_loop4 : cin number of stored data are read from local memory and then write to the output channel in a reverse order.

These four loops are used to make sure that the equivalence check can go through the loop bound limitation control. The local memory in the design is mapped to an external memory. The number of data that are read or write is controlled by cin in each loop. Although it is not possible to set cin to an infinitely big number, it is still reasonable to give a maximum value to cin according to the HW size. Several verification flows can be carried out with different values cin within the range HW size.

#### 3.4.2.2 Verification using SLEC

The C code is firstly used to generate HLS RTL using the Catapult. The local memory is set as an external memory, which prevents Catapult from generating HW registers for it. This provides the environment of memory access from the main functional block. A detailed Tcl script setting for Catapult is showed in Appendix 7.1.2.

In this example, there are two constraints need to be defined.

- ein : This is used for the control logic of output pattern. If ein is 2, there will be only one sequential output; otherwise there will be both sequential and reverse outputs. In this verification flow, ein is set to 2 for example.

- cin : This is used to control the number of data. The number of iteration time is depended on this variable. In this verification flow, cin is set to 16 for example. To make it safe, the constraints on the unroll_loop_with_limit is set to 17, which is 1 greater than cin.

The detailed way to define constraints files for CONS are showed in Appendix 7.1.3.

As is discussed in 3.4.1.2, a C-C verification was carried out first by invoking the Tcl file SLEC_compile_c_to_c.tcl which is modified according to section 3.4.1.2. The OOBA property check was turned on. The result can be seen in Appendix 0. All the properties were fully proven. There is no OOBA problem or falsification. This shows that the design is safe to the C-RTL verification flow.

After invoking the Tcl file SLEC_compile_rtl_v.tcl, the C-RTL verification also turned out to be successful. The result is shown in Appendix 0. Compared to the C-C result, C-RTL got fewer maps. This is because there is no property check or OOBA check in C-RTL verification flow.

More detail information is indicated in the corresponding log files which are stated in the result in 7.1.4.

#### 3.4.2.3 Summary

This part shows the way that SLEC actually work fine with variable loop bound. By using unroll_loop_with_limit constraints, loops with variable bound can be successfully verified. In the previous discussion, only 2 for ein and 16 for cin is showed as an example. More verification tests can be carried out by specifying other constraint numbers within the reasonable range of HW size. However, it is still a large effort if the design is very big with loop iteration. This is because too many different constraints have to be specified and verified to explicitly prove the design if the loop is not statically bounded.

As long as the design follows the guideline which has been mentioned in the summary of section 3.4.1.3, it will not contain the OOBA circumstance, and is safe to go through the SLEC verification flow.

# 4 Conclusion

In this report, several verification methodologies were discussed. A formal equivalence checker SLEC was specifically evaluated. This evaluation is based on the HLS flow. The outcome shows that SLEC can be used on constrained design specification.

## 4.1 Achievements

In automated HLS flow, SLEC can be safely applied when design specification satisfies:

- Untimed SLM
- Free of pointer alias
- Free of dynamic memory allocation
- Standard data types
- Standard channels
- Controllable loop bound

These limitations are based off of the SLEC tool capabilities as of summer 2013. They constraint the SLM to be very high-level abstracted. SystemC TLM needs to be modified to adapt to these guidelines.

TLM has different level of abstractions. With different timing details added in TLM, it can beyond the capability of SLEC to verify automatically. Manual timing specifications and constraints need to be defined to SLEC. This increases the verification effort.

In section 2.2.1, formal verification methods based on Bounded Model Checking (BMC) and System Verification Methodology (SVM) are introduced. These methods can be used for different abstraction level TLM verification.

In 2.2.2, Assertion Based Verification (ABV) with validation effort reuse tool (VERT) is introduced. This method can help to save the verification effort by reuse and transform the assertions and tests on TLM to RTL.

However, neither SVM nor ABV with use of VERT has been put into industry area. These researches still need to be proven in industry in the future.

With all the previous exploration, an overview of a new verification strategy is proposed below:

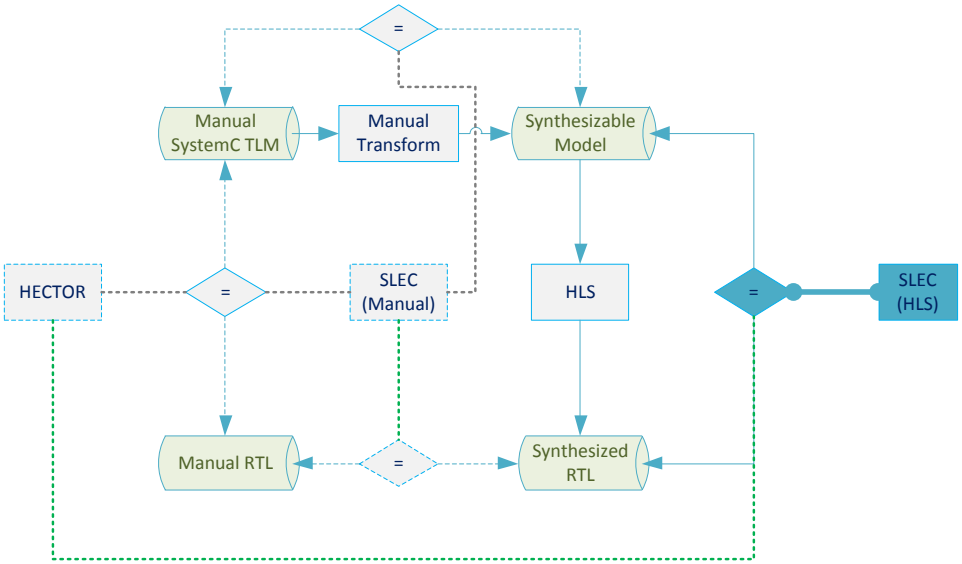


**Figure 21 Overview of New Verification Flow**

SLEC under HLS environment has been tested. It works under certain coding guideline mentioned before. However, it is only a small part to work out the through verification flow from the original TLM to the final HLS/manual RTL.

There are other methodologies described in section 2.2, but they are still in research stage and need time to be proven in industry area.

In the knowledge that SLEC for HLS is of the old style interface and technology, it is being improved. In the new release in later 2013, they remove the limitation around needing to constrain to limit the depth and variability of loops. And it can support malloc() system call providing it is used in a runtime static fashion. For example, if used in a C++ constructor SLEC will then be able to build the requisite memory. Clearly though it cannot be used in the truly dynamic sense that most software coders might use it, since the hardware cannot be dynamic.

## 4.2 Future Work

As is showed in Figure 21, this report has evaluated the SLEC for HLS. Those verificaiton method in dotted lines haven't been practised yet. Here are the suggestions for future work:

- Manually adaption of SLEC to different abstraction levels.

  This can be reasonably realized between different-way-generated RTL models either using black-box functional verification, or by manually adding detail HW specification to the I/O mapping. This is showed in green dotted line connected to SLEC (Manual) block.

  For the gray dotted lines connected to SLEC (Manual) block that indicate the verificaiton between different abstraction level design, it is possible when all the timing or structural detail are available and can be correctly mapped. It is not recommended to do so because of the complexity and the varity of design specification between original TLM/SLM and RTL. Lots of modification on the design are needed in order to use SLEC

- Test of HECTOR

  HECTOR can also be tested. Since HECTOR checks between C-models and RTL models, the most probable area is between synthesizable SLM in C and HLS generated RTL. It is easier to realize since the design specification are the same. This way is showed as green line connected to HECTOR block.

  For the gray dotted lin connected to HECTOR block, it is the same situation as that for SLEC, and is not recommended.

# 5 Supplementary Notes

This report is based on the technologies on Septmenber in 2013. The methods and tools involved are tightly connected to commertial issued agreed between Ericsson and Calypto (Mentor Graphics).

With the rapid speed of development in EDA vendor tool, the conclusion in this report is only for reference and more for thesis study used.

# 6 References


[1] L. Cai and D. Gajski, "Transaction level modeling: an overview," in *First IEEE/ACM/IFIP International Conference on Hardware/ Software Codesign and Systems Synthesis*, Newport Beach, CA, USA, 2003.

[2] "SoC Silicon and Software Design Cost Analysis: Costs for Higher Complexity Continue to Rise," May 2013. [Online]. Available: http://www.semico.com/content/soc-silicon-and-software-design-cost-analysis-costs-higher-complexity-continue-rise. [Accessed 2013].

[3] "IEEE Standard VHDL Language Reference Manual," *IEEE Std 1076-2008 (Revision of IEEE Std 1076-2002),* pp. c1-626, 03 February 2009.

[4] "IEEE Standard for SystemVerilog--Unified Hardware Design, Specification, and Verification Language," *IEEE Std 1800-2012 (Revision of IEEE Std 1800-2009),* pp. 1-1315, 22 February 2013.

[5] "IEEE Standard for the Functional Verification Language e," *IEEE Std 1647-2011 (Revision of IEEE Std 1647-2008),* pp. 1-495, 26 August 2011.

[6] "IEEE Standard for Property Specification Language (PSL)," *IEEE Std 1850-2010 (Revision of IEEE Std 1850-2005),* pp. 1-182, 6 April 2010.

[7] "IEEE Standard for Standard SystemC Language Reference Manual," *IEEE Std 1666-2011 (Revision of IEEE Std 1666-2005),* pp. 1-638, 09 January 2012.

[8] A. Namvar, "High Level Sysnthesis Evaluation of Tools and Methodology," Ericsson AB, Stockholm, 2013.

[9] "Universal Verification Methodology," Accellera Systems Initiative, [Online]. Available: http://www.accellera.org/community/uvm/.

[10] D. Große, H. M. Le and R. Drechsler, "Proving Transaction and System-level Properties of Untimed SystemC TLM Designs," in *Eighth ACM/IEEE International Conference on Formal Methods and Models for Codesign*, Grenoble, 2010.

[11] H. M. Le, D. Große and R. Drechsler, "Towards Analyzing Functional Coverage in SystemC TLM Property Checking," in *IEEE International Workshop on High-Level Design Validation and Test (HLDVT)*, Anaheim, 2010.

[12] M. F. S. Oliveira, C. Kuznik, W. Mueller, F. Haedicke, H. M. Le, D. Große, R. Drechsler, W. Ecker and V. Esen, "The System Verification Methodology for Advanced TLM Verification," in *International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS)*, Tampere, 2012.

[13] M. Chen and P. Mishra, "Assertion-Based Functional Consistency Checking between TLM and RTL Models," in *International Conference on VLSI Design*, Pune, 2013.

[14] B. Bailey, F. Balarin, M. McNamara, G. Mosenson, M. Stellfox and Y. Watanabe, TLM-Driven Design and Verification Methodology, San Jose: Cadence Design System, Inc., 2010.

[15] S. Vasudevan, V. Viswanath, J. A. Abraham and J. Tu, "Sequential Equivalence Checking between System Level and RTL Descriptions," *Design Automation for Embedded Systems,* pp. 377-396, 2008.

[16] Synopsys, "Synopsys HECTOR," Synopsys, Inc., 2013. [Online]. Available: http://www.synopsys.com/Tools/Verification/FunctionalVerification/Pages/hector.aspx. [Accessed 2013].

[17] "Sequential Equivalence Checking: A New Approach to Functional verification of Datapath and Control Logic Changes," Calypto Design Systems, Inc., 2012. [Online]. Available: http://calypto.com/en/page/leadform/11. [Accessed 2013].

[18] "RTL Verification Without Testbenches," Calypto Design Systems, Inc., 2012. [Online]. Available: http://calypto.com/en/page/leadform/13. [Accessed 2013].

[19] A. Mathur and V. Krishnaswamy, "Design for Verification in System-level Models and RTL," in *Design Automation Conference*, San Diego, 2007.

[20] F. Ghenassia, Transaction-Level Modeling with SystemC: TLM Concepts and Applications for Embedded Systems, Dordrecht: Springer, 2005.

[21] P. Georgelin and V. Krishnaswamy, "Towards A C++-based Design Methodology Facilitating Sequential Euivalence Checking," in *Design Automation Conference*, San Francisco, 2006.

[22] P. Chauhan, D. Goyal, G. Hasteer, A. Mathur and N. Sharma, "Non-cycle-accurate Sequential Equivalence Checking," in *Design Automation Conference*, San Francisco, 2009.

[23] "SLEC Product Family Datasheet," Calypto Design Systems, Inc., 2012. [Online]. Available: http://calypto.com/en/page/leadform/32. [Accessed 2013].

[24] "Cadence C-to-Silicon Compiler," Cadence Design Systems, Inc., [Online]. Available: http://www.cadence.com/products/sd/silicon_compiler/pages/default.aspx. [Accessed 2013].

[25] "Leveraging System Models for RTL Functional Verification using Sequential Logic Equivalence Checking," Calypto Design Systems, Inc., 2012. [Online]. Available: http://calypto.com/en/page/leadform/18. [Accessed 2013].

[26] M. Fingeroff, High-Level Synthesis Blue Book, Mentor Grphics, 2010.

# 7 Appendix

## 7.1 Simple Transaction Block with Memory via Channel

### 7.1.1 C Code

```
#include <ac_int.h>
#include <ac_channel.h>
#include <ac_fixed.h>

struct Control {
    ac_int<3, false> ein;
    ac_int<8, false> cin;
};

#pragma hls_design top
void test(ac_channel<ac_int<8> > &din, ac_channel<Control > &c_in,
ac_channel<ac_int<8> > &dout){

    static ac_int<8,false> mem[128];
    static bool initlflg = ac::init_array<AC_VAL_DC>(mem, 128); // Do not
initialize

    if (c_in.available(1) ) {
        Control current_control_in = c_in.read();
        if ( current_control_in.ein > 0) {

            count_loop1 : for(int i=0;i<current_control_in.cin;i++){
                mem[i] = din.read();
            }

            if ( current_control_in.ein == 2) {
                count_loop2 : for(int i=0;i<current_control_in.cin;i++){
                    dout.write(mem[i]);
                }
            } else {
                count_loop3 : for(int i=current_control_in.cin-1;i>=0;i--){
                    dout.write(mem[i]);
                }
```

```
                count_loop4 : for(int i=0;i<current_control_in.cin;i++){
                    dout.write(mem[i]);
                }
            }
        }
    }
}
```

### 7.1.2 Catapult SL Command Script

```
project new
flow package require /SLEC
flow package option set /SLEC/ENABLE_FLOP_MAPS true
flow package option set /SLEC/ENABLE_FULL_PROOF true
flow package option set /SLEC/GENERATE_HIERARCHICAL_PROOFS true
flow package option set /SLEC/MAXIMUM_AC_CHANNEL_SIZE 16
flow package require /SCVerify
solution file add ./test_be.cpp -type C++
solution file add ./testbench.cpp -type C++ -exclude true
directive set -REGISTER_IDLE_SIGNAL false
directive set -IDLE_SIGNAL {}
directive set -DONE_FLAG {}
directive set -START_FLAG {}
directive set -FSM_ENCODING none
directive set -REG_MAX_FANOUT 0
directive set -NO_X_ASSIGNMENTS true
directive set -SAFE_FSM false
directive set -RESET_CLEARS_ALL_REGS true
directive set -ASSIGN_OVERHEAD 0
directive set -DESIGN_GOAL area
directive set -OLD_SCHED false
directive set -CSA 0
directive set -TIMING_CHECKS true
directive set -MUXPATH true
directive set -REALLOC true
directive set -PIPELINE_RAMP_UP true
directive set -COMPGRADE fast
directive set -SPECULATE true
directive set -MERGEABLE true
directive set -REGISTER_THRESHOLD 256
directive set -MEM_MAP_THRESHOLD 32
directive set -UNROLL no
directive set -CLOCK_OVERHEAD 20.000000
directive set -OPT_CONST_MULTS -1
go analyze
directive set -TRANSACTION_DONE_SIGNAL true
directive set -CLOCK_NAME clk
directive set -CLOCKS {clk {-CLOCK_PERIOD 5.0 -CLOCK_EDGE rising -
CLOCK_UNCERTAINTY 0.0 -CLOCK_HIGH_TIME 2.5 -RESET_SYNC_NAME rst -
```

```
RESET_ASYNC_NAME arst_n -RESET_KIND sync -RESET_SYNC_ACTIVE high -
RESET_ASYNC_ACTIVE low -ENABLE_NAME {} -ENABLE_ACTIVE high}}
directive set -TECHLIBS {{mgc_sample-065nm-dw_beh_dc.lib mgc_sample-065nm-
dw_beh_dc} {ram_sample-065nm-singleport_beh_dc.lib ram_sample-065nm-
singleport_beh_dc}}
directive set -DESIGN_HIERARCHY
test__FR52ac_channel__tm__34_31ac_int__tm__17_XCiL_1_8XCbL_1_1R26ac_channel
__tm__9_7ControlT1
go compile
directive set /test/core/mem:rsc -EXTERNAL_MEMORY true
go architect
go extract
```

### 7.1.3 Custom Constraints Settings

slec_pre_build.tcl:

```
unroll_loop_with_limit -spec -file ../../../test_be.cpp -label count_loop1
-limit 17
unroll_loop_with_limit -spec -file ../../../test_be.cpp -label count_loop2
-limit 17
unroll_loop_with_limit -spec -file ../../../test_be.cpp -label count_loop3
-limit 17
unroll_loop_with_limit -spec -file ../../../test_be.cpp -label count_loop4
-limit 17

unroll_loop_with_limit -impl -file ../../../test_be.cpp -label count_loop1
-limit 17
unroll_loop_with_limit -impl -file ../../../test_be.cpp -label count_loop2
-limit 17
unroll_loop_with_limit -impl -file ../../../test_be.cpp -label count_loop3
-limit 17
unroll_loop_with_limit -impl -file ../../../test_be.cpp -label count_loop4
-limit 17
```

slec_contraints.tcl:

```
set ein 3'd1
set cin 8'd16

set_constant -spec -module spec_wrapper -value $ein
slecIN_c_in_ein_rsc_idx_0
set_constant -spec -module spec_wrapper -value $cin
slecIN_c_in_cin_rsc_idx_0

set_constant -impl -module impl_wrapper -value $ein
slecIN_c_in_ein_rsc_idx_0
set_constant -impl -module impl_wrapper -value $cin
slecIN_c_in_cin_rsc_idx_0
```

### 7.1.4 SLEC Result

C-RTL

```
Summary of key results:
================================================================================
**************** Design Rule Violations (.cat2slec/ccheck.log) ****************
================================================================================
                   Errors   Warnings
Spec                    0         49
Impl                    0          0
================================================================================
*********** Design Characteristics (.cat2slec/characteristics.log) ************
================================================================================
    Inputs/BBox-outputs   Outputs/BBox-Inputs    Inouts    WL-Flops    BL-Flops
Spec                 20                    32         0          57         414
Impl                 24                    32         0         179        1382
================================================================================
********************** Mapping (.cat2slec/mapping.log) **********************
================================================================================
     Input-Maps     Output-Maps         Active Intermediate-Maps
             16              38                                1
     Inactive Intermediate-Maps     Unreachable Input-Maps     Memory-maps
                              0                          0               0
     Unmapped Inputs      Unmapped Outputs      Unmapped Flops
Spec               4                     0                  57
Impl               8                     0                 178
================================================================================
************************** Reset (.cat2slec/reset.log) **********************
================================================================================
           Reset WL-Flops      Unreset WL-Flops
Spec                   57                    0
Impl                  179                    0
================================================================================
***************** Equivalence Results (.cat2slec/results.log) *****************
================================================================================
                       Proven Cond-Proven Bounded-Proven Falsified Unresolved
Output-Maps                38           0              0         0          0
Active Intermediate-Maps    0           0              0         0          0
Properties                  0           0              0         0          0
```

C-C (Against Itself)

```
Summary of key results:
================================================================================
****************** Design Rule Violations (calypto/ccheck.log) ****************
================================================================================
                   Errors   Warnings
Spec                    0        769
Impl                    0        769
================================================================================
************* Design Characteristics (calypto/characteristics.log) ************
================================================================================
    Inputs/BBox-outputs   Outputs/BBox-Inputs    Inouts    WL-Flops   BL-Flops
Spec                515                   512         0         586       4408
Impl                515                   513         0         553       4375
================================================================================
************************** Mapping (calypto/mapping.log) **********************
================================================================================
Input-Maps     Output-Maps         Active Intermediate-Maps
        34             524                              553
Inactive Intermediate-Maps     Unreachable Input-Maps     Memory-maps
                         0                        478               0
     Unmapped Inputs      Unmapped Outputs      Unmapped Flops
Spec               3                     0                  33
Impl               3                     0                   0
================================================================================
***************************** Reset (calypto/reset.log) **********************
================================================================================
           Reset WL-Flops      Unreset WL-Flops
Spec                  586                    0
Impl                  553                    0
================================================================================
****************** Equivalence Results (calypto/results.log) *****************
================================================================================
                      Proven Cond-Proven Bounded-Proven Falsified Unresolved
Output-Maps              524           0              0         0          0
Active Intermediate-Maps  39           0              0         0          0
Properties                33           0              0         0          0
```